Imperial College London

Master's Thesis

Department of Computing

Imperial College of Science, Technology and Medicine

# Classical Call By Push Value

*Author:*
Emma Tye

*Supervisor:*
Steffen van Bakel

June 17, 2020

Submitted in partial fulfillment of the requirements for the Maths and Computer Science MEng of Imperial College London

**Abstract**

Call by Push Value (CBPV) is a calculus for explicitly modelling call-by-name and call-by-value reduction in the $\lambda$-calculus and many other intuitionistic logic-based calculi. We investigate whether classical logic-based calculi, specifically the $\bar{\lambda}\mu\tilde{\mu}$-calculus, can model CBPV and therefore subsume the call-by-name and call-by-value paradigms.

We augment the $\bar{\lambda}\mu\tilde{\mu}$-calculus syntax with new features inspired by CBPV, and provide a translation from CBPV into the augmented $\bar{\lambda}\mu\tilde{\mu}$-calculus. By limiting the traditional reduction of $\bar{\lambda}\mu\tilde{\mu}$, we can simulate CBPV reduction in this augmented calculus. This should open up avenues for describing the semantics of call-by-name and call-by-value via classical logic semantics.

# Contents

# Chapter 1

# Introduction

The call-by-name and call-by-value subsets of the $\lambda$-calculus are important paradigms both in theory and in practice. Classical logic has been shown to represent an abstract machine which models both of these paradigms [1], and the calculus Call by Push Value [2] was intentionally constructed to explicitly model these reduction strategies. In particular, the interpretations of call-by-name and call-by-value $\lambda$-calculus into Call by Push Value preserves the semantics of the respecitve subsets. A link between these two concepts - classical logic and Call by Push Value - would open interesting discussion into the semantics of these calculi, and what role effectful computation has in classical logic.

## 1.1 Call by Push Value

Call by Push Value (CBPV) [2] was introduced as a subsuming paradigm for the call-by-name and call-by-value paradigms of the $\lambda$-calculus. The reduction order of a CBPV computation is deterministic, and can be thought of as encoded in the term itself. The call-by-name $\lambda$-calculus and call-by-value $\lambda$-calculus are interpreted differently into CBPV, since their reduction order is different. The benefit to this is that their interpretations preserve their operational and denotational semantics, so if you were to introduce a new set of semantics for call-by-name and call-by-value reduction, you only need to define it for CBPV. Another benefit is if you introduce a new intuitionistic logic-based calculi, then you only need to provide an interpretation in CBPV for it to inherit the semantics that have been provided for CBPV.

CBPV splits its terms into distinct categories: computations and values. The slogan from [2] is

> A value is, a compuation does.

Computations reduce to each other, and values are substituted or applied. A computation can be used as a value if it is "thunked", and a value can (try) be used as a computation if it is "forced". This arbitrarily halts and continues computation. Other computations include $\lambda$ abstractions (contrary to the definition of values and terms in call-by-value $\lambda$-calculus), applications, sequencing and returning.

It's type system has a Curry-Howard-style correspondence with intuitionistic linear logic [3], hence only intuitionistic calculi can be translated into CBPV.

In this paper, we consider pure, functional CBPV without effects - in particular, we do not the consider sum and product types that were included in the original definition of CBPV.

## 1.2 $\bar{\lambda}\mu\tilde{\mu}$-calculus

The $\bar{\lambda}\mu\tilde{\mu}$-calculus [1] has a Curry-Howard-style correspondence with classical logic, in particular sequent calculus-style classical logic. It arose from the $\lambda\mu$-calculus [4] and the call-by-name and call-by-value reduction strategies of the $\lambda$-calculus. It can be seen as an abstract machine in which to run the different strategies of the $\lambda$-caluclus [5], and is therefore ideal for modelling CBPV.

There are three distinct categories of terms in $\bar{\lambda}\mu\tilde{\mu}$ - values, contexts and commands. From the perspective of an abstract machine, a value can be thought of as a $\lambda$-calculus term, a context as a stack of $\lambda$-calculus values, and a command as a machine. Commands contain a single value and context, and attempts to "run the value in the context". However, it tries to do this with only variables, stacks and various function symbols.

For example, when running the $\lambda$-calculus in a call-by-name abstract machine, the term $\lambda x.s$ takes a term off the stack and substitutes it for $x$ in $s$. But $st$ is not a function or a variable, but it does interact with the stack. In both call-by-name and call-by-value abstract machines, it pushes $t$ onto the stack and then attempts to run $s$ to a term of the form $\lambda x.s'$. So we can think of it as a function that takes some existing stack, and returns the machine which runs $s$ in the existing stack with $t$ pushed on top. There are similar ideas for constructing the rest of the language and its reduction rules.

The $\mathcal{X}$-calculus [6] could also have been chosen for this paper. It also has a Curry-Howard correspondence with sequent calculus-style classical logic, and arguably models the logic closer than $\bar{\lambda}\mu\tilde{\mu}$ does. Each reduction in the $\mathcal{X}$-calculus explicitly models a cut reduction (see Section 2.3), and it's typing rules are isomorphic to the inference rules of classical sequent calculus. In $\bar{\lambda}\mu\tilde{\mu}$, certain terms represent the inference rules of classical sequent calculus, but not the original constructs. $\bar{\lambda}\mu\tilde{\mu}$ can be thought of as a "focused" version of sequent calculus, where in the sequent $\Gamma \vdash \Sigma$, at most one formula is focused on (and is active in whatever inference rule is being applied).

There is a cut reduction that $\bar{\lambda}\mu\tilde{\mu}$ does not model, however [6, page 37]. In the future, the reduction of $\bar{\lambda}\mu\tilde{\mu}$ might include this additional reduction, or it might be useful to provide our interpretation of CBPV into the $\mathcal{X}$-calculus instead.

## 1.3 An interpretaiton

We give an interpretation of CBPV into $\bar{\lambda}\mu\tilde{\mu}$ in Chapter 9, based on the behaviour of CBPV in an abstract machine [7]. Using the motivation for $\bar{\lambda}\mu\tilde{\mu}$ as an abstract machine to run the $\lambda$-calculus in, we attempt to use a similar technique for translating CBPV. However, the artificial halting and continuing of computations - "thunking" and "forcing" - cannot be modelled by the traditional $\bar{\lambda}\mu\tilde{\mu}$ syntax. A computation $M$ that is "thunked" and then immediately "forced" continues running as $M$, removing the surrounding "thunk" and "force" syntax. In an abstract machine, this computation doesn't interact with the stack at all, but still reduces. So it cannot be represented as a function or a variable, hence cannot be translated into $\bar{\lambda}\mu\tilde{\mu}$ syntax.

To overcome this, we augment the syntax of $\bar{\lambda}\mu\tilde{\mu}$ to include "thunking" and "forcing" of values, with the canonical reduction from CBPV. We also present a restriction of this augmented $\bar{\lambda}\mu\tilde{\mu}$-calculus' reductions in order to simulate CBPV reduction.

# Chapter 2

# Intuitionistic and Classical Logic

Classical logic is the most common type of logic used in mathematics, where every formula has a binary truth value. Intuitionistic logic is a subset of classical logic, where we can no longer assume any formula that is not "true" is "false" - i.e. $\neg\neg A \not\equiv A$. We will refer to the property that $\neg\neg A \equiv A$ as the law of the excluded middle.

Natural deduction and sequent calculus are formal deduction systems for intuitionistic and classical logic, defined by Gentzen in 1935 [8]. Natural deduction was designed to model hand-written proofs of logical formulae, whereas sequent calculus proofs can be constructed from the bottom up. Both systems are equivalent: a formula is provable in one system if and only if it is provable in the other.

In both classical and intutitionistic propositional logic, the set $\{\rightarrow, \bot\}$ is adequate (every propositional formula can be expressed using only these connectives/elements). In fact, we define $\neg A := A \rightarrow \bot$. This is called implicational propositional logic, and we will only work with these connectives in this paper.

## 2.1 Natural deduction

In natural deduction, a sequent $\Gamma \vdash A$ says that "if all assumptions in the set $\Gamma$ hold, then $A$ follows". This differs from Gentzen's definition of natural deduction by keeping a record of all assumed formulae in the proof so far, but follows the format of type judgements for the $\lambda$-calculus (and $\lambda\mu$-calculus).

*Notation.*

- We use $\Gamma$ as a set of formulae.
- We use $\Gamma, A$ (or equivalently $A, \Gamma$) to mean the set $\Gamma \cup \{A\}$.

**Definition 2.1** (Natural deduction)**.** Our definition of natural deduction for (implicational) intuitionistic propositional logic is:

$$(\text{if } A \in \Gamma)\ \frac{}{\Gamma \vdash A}\ (\text{Ax})$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ (\rightarrow I)$$

$$\frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\rightarrow E)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(\bot E)$$

For classical logic (for propositional, remove the predicate-specific rules), we add the following rule:

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,(\neg\neg E)$$

where $\neg A = A \to \bot$.

Using natural deduction, we can construct proof-trees for formulae. Valid formulae correspond to formulae that can be proven with no assumptions (i.e. the final sequent is of the form $\emptyset \vdash A$).

**Example.** Let $\Gamma = (A \to B) \to A$

A proof of Pierce's law $((A \to B) \to A) \to A$ can be constructed:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \neg A \vdash (A \to B) \to A}\,(\text{Ax}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \neg A, A \vdash A}\,(\text{Ax}) \quad \dfrac{}{\Gamma, \neg A, A \vdash \neg A}\,(\text{Ax})}{\Gamma, \neg A, A \vdash \bot}\,(\to E)}{\Gamma, \neg A, A \vdash B}\,(\bot E)}{\Gamma, \neg A \vdash A \to B}\,(\to I)}{\Gamma, \neg A \vdash A}\,(\to E) \qquad \dfrac{}{\Gamma, \neg A \vdash \neg A}\,(\text{Ax})}{\Gamma, \neg A \vdash \bot}\,(\to E)}{\Gamma \vdash \neg\neg A}\,(\to I)}{\Gamma \vdash A}\,(\neg\neg E)}{\vdash ((A \to B) \to A) \to A}\,(\to I)$$

Notice that the rule $(\neg\neg E)$ must be used in this proof, highlighting the fact that Pierce's Law is only valid in classical logic.

## 2.2 Sequent calculus

In sequent calclulus, a sequent $\Gamma \vdash \Sigma$ can be read as: the conjunction of all formulae in $\Gamma$ implies the disjunction of all formulae in $\Sigma$.

*Notation.*

In the sequent $\Gamma \vdash \Sigma$, we call $\Gamma$ the antecedent and $\Sigma$ the succedent.

In the rule

$$\frac{\Gamma_1 \vdash \Sigma_1 \qquad \ldots \qquad \Gamma_n \vdash \Sigma_n}{\Delta \vdash \Lambda}$$

we call $\Gamma_i \vdash \Sigma_i$ the upper sequents and $\Delta \vdash \Lambda$ the lower sequent.

All other notation is carried forward from Section 2.1. Note that in sequent calculus the $\Gamma$ and $\Sigma$ are ordered, which is why the rule (Interchange) is introduced.

**Definition 2.2** (Sequent calculus)**.** The rules of sequent calculus are defined as:

$$\frac{\Gamma \vdash \Sigma, A \qquad A, \Gamma \vdash \Sigma}{\Gamma \vdash \Sigma}\ (\text{Cut})$$

$$\frac{}{\Gamma, A \vdash A, \Sigma}\ (\text{Ax})$$

$$\frac{A, \Gamma \vdash \Sigma, B}{\Gamma \vdash \Sigma, A \to B}\ (\to IS)$$

$$\frac{\Gamma \vdash \Sigma, A \qquad B, \Gamma \vdash \Sigma}{A \to B, \Gamma \vdash \Sigma}\ (\to IA)$$

$$\frac{}{\bot, \Gamma \vdash \Sigma}\ (\bot IA)$$

Intuitionistic logic has the restriction that at most one formula is allowed in the succedent.

Notice that in all the rules, except for (Cut), every formula that appears in an upper sequent is a subformula of some formula in the lower sequent. We call this the *subformula* property. Natural deduction does not have this property, for example the ($\to$ E) rule.

An extra rule, *weakening*, may be added, where formulas can be added to either the antecedent or the succedent. We use this rule implicitly, as any proof of $\Gamma \vdash \Sigma$ can also be used to prove $A, \Gamma \vdash \Sigma, B$.

Using sequent calculus we can also construct proof-trees for formulae.

**Example.** A proof of Pierce's law $((A \to B) \to A) \to A$ can be constructed:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A, B}\ (\text{Ax})}{\vdash A, A \to B}\ (\to \text{IS}) \qquad \dfrac{}{A \vdash A}\ (\text{Ax})}{(A \to B) \to A \vdash A}\ (\to \text{IA})}{\vdash ((A \to B) \to A) \to A}\ (\to \text{IS})$$

## 2.3 Cut reductions

One of the main benefits of sequent calculus over natural deduction is that most of its rules have the subformula property. This makes it easy to construct proofs from the bottom up, as no new formulas are needed to be introduced. In fact, for each formula there is only one rule (other than (Cut)) that it can be active in. So ideally the (Cut) rule would be unnecessary.

In 1935, Gentzen proved that all (Cut) rules can be eliminated. They proved that sequent calculus proofs can undergo *cut elimination*, and that this process terminates, resulting in proofs in a "normal form" that had no usage of the (Cut) rule.

**Theorem 2.3** (Hauptsatz)**.** *Any sequent calculus proof of a formula $A$ can be reduced to a proof of $A$ that doesn't contain the rule (Cut).*

*Proof.*

This proof starts with the uppermost (Cut) rule - a (Cut) rule that has no (Cut) rules above it. It then proves that the lower sequent of the (Cut) can be proved without the need for (Cut) rules. Applied repeatedly, this removes (Cut) rules from a proof.

Consider a derivation of the form:

$$\frac{\Gamma \vdash \Sigma, A \qquad A, \Gamma \vdash \Sigma}{\Gamma \vdash \Sigma}\,(\text{Cut})$$

We will call $A$ the cut formula.

The proof uses two kinds of induction - induction on the *degree* of a derivation, and induction on the *rank* of a derivation.

The *degree* of a derivation is the degree of the cut formula (i.e. the number of logical symbols occuring in the formula).

The *rank* of a derivation is the sum of the left rank and right rank of the derivation. The left rank is the largest consecutive number of sequents on the upper-left-hand side of the (Cut) rule that contain the cut formula in the succedent. The right rank is the largest consecutive number of sequents on the upper-right-hand side of the (Cut) rule that contain the cut formula in the antecedent.

For example, the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B, A, B \vdash B, C}}{A \to B, A \vdash B, B \to C}}{A \to B \vdash A \to B, B \to C} \qquad \dfrac{}{B \to C, A \to B \vdash A \to B}}{A \to B \vdash A \to B}$$

has degree 1, and left rank 2, right rank 1 so overall rank 3 (the cut formula is $B \to C$).

For simplicity, we assume that $\bot \notin \Gamma$ (otherwise the sequent $\Gamma \vdash \Sigma$ can be proved by simply ($\bot$ IA)), that $\Gamma$ and $\Sigma$ share no formulas (otherwise the sequent $\Gamma \vdash \Sigma$ can be proved by simply (Ax)) and that the cut formula doesn't appear in either $\Gamma$ or $\Sigma$ (otherwise one of the upper sequents of the (Cut) rule would be the sequent $\Gamma \vdash \Sigma$).

- Base case: rank = 2

  If the rank is 2, then the cut formula must be introduced immediately on both sides of the cut (as we've ruled out other cases which would cause the rank to be 1 on either side).

  - Base case: degree = 0

    If the degree of a formula is 0 (i.e. has no $\to$ connective) then it must be introduced via some (Ax) rule. The derivation looks like:

    $$\frac{\dfrac{}{\Gamma \vdash \Sigma, A} \qquad \dfrac{}{A, \Gamma \vdash \Sigma}}{\Gamma \vdash \Sigma}$$

    Hence $A \in \Gamma$ and $A \in \Sigma$. But we previously assumed that $\Gamma$ and $\Sigma$ were distinct, otherwise $\Gamma \vdash \Sigma$ could be proved via (Ax), so we are done.

– Inductive case: reduction of degree

Say the cut formula is $A \to B$. It must be introduced via ($\to$ IS) on the left-hand-side and ($\to$ IA) on the right-hand-side. The derivation looks like:

$$\dfrac{\dfrac{\Gamma, A \vdash \Sigma, B}{\Gamma \vdash \Sigma, A \to B} \quad \dfrac{\Gamma \vdash \Sigma, A \qquad B, \Gamma \vdash \Sigma}{A \to B, \Gamma \vdash \Sigma}}{\Gamma \vdash \Sigma}$$

this can be re-arranged to give:

$$\dfrac{\dfrac{\Gamma \vdash \Sigma, A \qquad \Gamma, A \vdash \Sigma, B}{\Gamma \vdash \Sigma, B} \qquad B, \Gamma \vdash \Sigma}{\Gamma \vdash \Sigma}$$

The degree of the cut formula(s) has reduced, and so by induction this proof is cut-free.

However, this proof could be re-arranged in a different way:

$$\dfrac{\Gamma \vdash \Sigma, A \qquad \dfrac{\Gamma, A \vdash \Sigma, B \qquad B, \Gamma \vdash \Sigma}{\Gamma, A \vdash \Sigma}}{\Gamma \vdash \Sigma}$$

This causes cut-elimination to be non-confluent, as these are completely different proofs.

- Inductive case: reduction of rank

There are two cases: when the left rank is greater than 1, and when the right rank is greater than 1.

– Left rank is greater than 1

The derivation looks like:

$$\dfrac{\dfrac{\dots}{\Gamma \vdash \Sigma, A}\,(?) \qquad A, \Gamma \vdash \Sigma}{\Gamma \vdash \Sigma}$$

The rule (?) must either be ($\to$ IA) or ($\to$ IS), as the other two rules end immediately and would cause the rank to be 1.

If (?) is ($\to$ IA), then the derivation looks like:

$$\dfrac{\dfrac{\Gamma' \vdash \Sigma, B, A \qquad C, \Gamma' \vdash \Sigma, A}{B \to C, \Gamma' \vdash \Sigma, A} \qquad A, B \to C, \Gamma' \vdash \Sigma}{B \to C, \Gamma' \vdash \Sigma}$$

where $\Gamma' = \Gamma \setminus \{B \to C\}$. This can be re-arranged to give:

$$\dfrac{\dfrac{\Gamma' \vdash \Sigma, B, A \qquad A, B \to C, \Gamma' \vdash \Sigma}{B \to C, \Gamma' \vdash \Sigma} \quad \dfrac{C, \Gamma' \vdash \Sigma, A \qquad A, B \to C, \Gamma' \vdash \Sigma}{C, B \to C, \Gamma' \vdash \Sigma}}{B \to C, \Gamma' \vdash \Sigma}$$

If (?) is ($\to$ IS), then the derivation looks like:

$$\dfrac{\dfrac{\Gamma, B \vdash C, \Sigma', A}{\Gamma \vdash B \to C, A, \Sigma'} \qquad A, \Gamma \vdash B \to C, \Sigma'}{\Gamma \vdash B \to C, \Sigma'}$$

where $\Sigma' = \Sigma \setminus \{B \to C\}$. This can be re-arranged to give:

$$\dfrac{\dfrac{\Gamma, B \vdash C, \Sigma', A \qquad A, \Gamma \vdash B \to C, \Sigma'}{\Gamma, B \vdash C, B \to C, \Sigma'}}{\Gamma \vdash B \to C, \Sigma'}$$

The left rank is reduced, so by induction this proof is cut-free.

– Right rank is greater than 1
Symmtrical to the left rank proof.

See [8], Section 3: "Proof of the Hauptsatz" for the full proof, including the other connectives that we ignore in this chapter. □

# Chapter 3

# $\lambda$ calculus

The lambda calculus, or $\lambda$-calculus, was developed by Church in 1936 [9]. It was originally constructed as a way to represent all computable functions, but we will be examining it from a different perspective: as a representation of implicational intuitionistic logic proofs via natural deduction (see Section 2.1). It's simple type system is isomorphic to intuitionistic logic, and the terms themselves represent proofs. This isomorphism is known as the Curry-Howard isomorphism [10].

## 3.1 Lambda calculus overview

The lambda calculus is made up of terms - each term corresponds to a program, or computable function. There are three main types of terms: a variable, a function (with parameter) and function application.

**Definition 3.1** ($\lambda$ calculus syntax)**.** Lambda calculus terms are defined as follows:

$$s, t ::= x \mid \lambda x.s \mid st$$

We call smaller terms within a term its subterms.

**Definition 3.2** ($\lambda$ calculus subterms)**.** Let $s$ be any $\lambda$-calculus term.

- $s$ is a subterm of $s$.
- If $s = x$, then there are no other subterms of $s$.
- If $s = \lambda x.s'$, then the subterms of $s'$ are also subterms of $s$.
- If $s = s't'$, then the subterms of $s'$ and $t'$ are also subterms of $s$.

There are no other subterms of $s$.

A subterm of $s$ that is not $s$ itself is a proper subterm of $s$.

The variables within terms fall into two categories: bound and free. Bound variables are all variables which have a corresponding lambda abstraction surrounding it. Free variables are variables that are not bound.

**Definition 3.3** (Free and bound variables)**.** The set of free variables of a term $s$, denoted $\text{free}(s)$, is defined recursively as follows:

$$\begin{aligned} \text{free}(x) &= \{x\} \\ \text{free}(\lambda x.s) &= \text{free}(s) \setminus \{x\} \\ \text{free}(st) &= \text{free}(s) \cup \text{free}(s) \end{aligned}$$

The set of bound variables of a term $s$, denoted $\text{bound}(s)$, is defined recursively as follows:

$$\begin{aligned} \text{bound}(x) &= \emptyset \\ \text{bound}(\lambda x.s) &= \text{bound}(s) \cup \{x\} \\ \text{bound}(st) &= \text{bound}(s) \cup \text{bound}(t) \end{aligned}$$

We define the set of all variables in a term as

$$\text{variables}(s) = \text{free}(s) \cup \text{bound}(s)$$

**Definition 3.4** (Closed terms)**.** A closed term $s$ is a term with no free variables - i.e. $\text{free}(s) = \emptyset$.

A change of bound variable is called alpha conversion, and two terms that can be converted to one another via alpha conversion are called alpha equivalent ($=^{\alpha}$).

The only notion of reduction in the $\lambda$-calculus is function application, called $\beta$-reduction. This uses substitution to replace an input variable by an applied term.

**Definition 3.5** (Term substitution)**.** Term substitution is a recursive operation on terms defined as follows:

$$\begin{aligned} x\,[t/x] &= t && \\ y\,[t/x] &= y && y \neq x \\ (\lambda x.s)\,[t/x] &= \lambda x.s && \\ (\lambda y.s)\,[t/x] &= \lambda z.(s\,[y/z]\,[t/x]) && y \neq x, z \notin \text{free}(s) \cup \text{free}(t) \\ (s_1 s_2)\,[t/x] &= (s_1\,[t/x])(s_2\,[t/x]) && \end{aligned}$$

**Definition 3.6** ($\beta$-reduction)**.** The primitive $\beta$-reduction rule is defined as:

$$(\lambda x.s)t \rightsquigarrow_{\beta} s\,[t/x]$$

$\beta$-reduction can occur in any subterm of a term.

*Notation.* We use $\rightsquigarrow_{\beta}^{*}$ to mean the transitive closure of $\rightsquigarrow_{\beta}$. In general, for any reduction relation $\rightsquigarrow$, $\rightsquigarrow^{*}$ represents its transitive closure.

We will normally reserve $\rightsquigarrow_{\beta}$ for primitive $\beta$-reduction, or to distinguish between other kinds of similar reduction. When the reduction rule used is unambiguous, or is unimportant, we will use $\rightsquigarrow$.

**Example.**

$$\begin{aligned} y((\lambda x.x)z) &\rightsquigarrow yz \\ (\lambda x.xx)(\lambda x.xx) &\rightsquigarrow (\lambda x.xx)(\lambda x.xx) \end{aligned}$$

A term that can be reduced is called a redex. Terms that cannot be reduces are called values and are said to be in normal form.

**Definition 3.7** (Normal form)**.** A term in normal form is defined as follows:

$$N ::= x \mid \lambda x.N \mid xN_1...N_n$$

Some terms eventually reduce to normal forms:

**Definition 3.8** (Weakly normalising terms)**.** A term $s$ is weakly normalising if there exists some term $t$ such that $s \rightsquigarrow^* t$ and $t$ is in normal form.

And some terms always reduce to a normal form:

**Definition 3.9** (Strongly normalising terms)**.** A term $s$ is strongly normalising if all reductions on $s$ eventually reach a normal form (i.e. there are no infinite reduction sequences for $s$).

Not all terms are strongly or even weakly normalising, however.

**Example.** Consider the term $(\lambda x.xx)(\lambda x.xx)$ (note that it is not in normal form). This term can reduce to itself in one $\beta$-reduction step, so it is not strongly normalising. But there is no other way to reduce the term (none of its subterms are redexes), so it is not even weakly normalising.

$\beta$-reduction has an important property:

**Definition 3.10** (Confluence)**.** A relation on terms $\rightsquigarrow$ is confluent if and only if the following property holds for all terms $s$:

$$s \rightsquigarrow t_1 \text{ and } s \rightsquigarrow t_2 \implies$$
$$\exists t.\ t_1 \rightsquigarrow^* t \text{ and } t_2 \rightsquigarrow^* t$$

**Proposition 3.11** ($\beta$-reduction is confluent)**.** The reduction relation $\rightsquigarrow$ is confluent.

This property implies that if $s$ $\beta$-reduces to a normal form, that is the only normal form $s$ could reduce to.

**Proposition 3.12** (Uniqueness of normal forms)**.** If $s$ is weakly normalising under $\beta$-reduction, and reduces to normal forms $t_1$ and $t_2$ then $t_1 =^\alpha t_2$.

## 3.2 Simply typed lambda calculus

Church gave his simple type system for the $\lambda$-calculus in [11]. It was meant to ensure that programs were well-formed - functions occurred on the left-hand-side of an application. There exist terms which cannot be typed, for example $(\lambda x.xx)(\lambda x.xx)$. We already saw that this term reduced infinitely - the simple type system prevents anomalies such as this and ensures that terms always reduce to a normal form (are strongly normalising 3.9).

The simple type syntax for the $\lambda$-calculus consists of basic types ($\phi$) and the binary connective $\rightarrow$.

**Definition 3.13** ($\lambda$-calculus type syntax)**.** $\lambda$-calculus types are defined as:

$$A, B ::= \phi \mid A \rightarrow B$$

*Notation.* The type connective $\rightarrow$ is right-associative, so $A \rightarrow B \rightarrow C$ is read as $A \rightarrow (B \rightarrow C)$.

We can judge whether a term has a certain type, given a partial map of variables to types.

*Notation.* We take $\Gamma$ to be a partial map of variables to types. We use $\Gamma, x : A$, where $x \notin \Gamma$ to mean the partial function $\Gamma$ with the additional map of $x \mapsto a$.

**Definition 3.14** ($\lambda$-calculus type assignments)**.** A typing judgement for a term has the form $\Gamma \vdash s : A$. Type assignemnts are:

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \to B}$$

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B}$$

*Notation.* If $\emptyset \vdash s : A$, then we say that $s$ has type $A$, or just $s : A$.

A term may have more than one type.

**Example.** Consider $\lambda x.x$. It has type $\phi \to \phi$:

$$\frac{\dfrac{}{x : \phi \vdash x : \phi}}{\emptyset \vdash \lambda x.x : \phi \to \phi}$$

However it also has type $(\phi \to \phi) \to \phi \to \phi$:

$$\frac{\dfrac{}{x : \phi \to \phi \vdash x : \phi \to \phi}}{\emptyset \vdash \lambda x.x : (\phi \to \phi) \to \phi \to \phi}$$

It may not be possible to type a term.

**Example.** Consider $(\lambda x.xx)(\lambda x.xx)$. Say it has type $B$.

The only rule for a term of the form $st$ is:

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash st : B}$$

So we must have that $(\lambda x.xx)$ can be typed with both $A \to B$ and $A$, for some $A$. Let's try to prove $\lambda x.xx : A \to B$ for some $A$.

The only rule for a term of the form $\lambda x.s$ is:

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \to B}$$

So we must have that $x : A \vdash xx : B$. Since $xx$ is of the form $st$, we must have that $x : A \vdash x : C \to B$ and $x : A \vdash x : C$. The only rule for the term $x$ is:

$$\frac{}{\Gamma, x : A \vdash x : A}$$

So we must have that $A = C \to B$ and $A = C$, which is impossible.

Hence we cannot type $(\lambda x.xx)(\lambda x.xx) : B$ for any type $B$, and so it is untypeable.

In fact, every typeable term is strongly normalising.

**Lemma 3.15** (Typeable terms are strongly normalising)**.** *Let $s$ be a $\lambda$-calculus term. If $\Gamma \vdash s : A$ is a valid type judgement, for some $\Gamma$ and some $A$, then $s$ is strongly normalising.*

Types are also preserved through reduction.

**Lemma 3.16** (Type preservation through reduction)**.** *Let $s, s'$ be $\lambda$-calculus terms such that $s \rightsquigarrow s'$. If $\Gamma \vdash s : A$, then $\Gamma \vdash s' : A$.*

## 3.3 Curry-Howard isomorphism

There are syntactic similarities between natural deduction (see Section 2.1) and the simple type system of the $\lambda$-calculus. For example, the natural deduction rule ($\to$ I):

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

and the typing rule for the term $\lambda x.s$:

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x.s : A \to B}$$

In fact, each construction in the $\lambda$-calculus provides a "witness" to a certain natural deduction rule. $x$ is a witness for (Ax), $\lambda x.s$ is a witness for ($\to$ I) and $st$ is a witness for ($\to$ E). Using this, we can translate (typed) $\lambda$-calculus terms to natural deduction proofs and vice versa. Notice that there is no $\lambda$-calculus construct or typing rule corresponding to ($\neg\neg$ E), so the proofs that the $\lambda$-calculus witnesses are intuitionistic proofs. There is also no term to witness ($\bot$ E).

Curry and Howard discovered this correspondence, and Howard formalised their work in 1969 [10].

**Theorem 3.17** (Curry-Howard isomorphism)**.** *There exists a natural deduction proof of the intuitionistc sequent $\Gamma \vdash A$ without use of ($\bot$ E) if and only if there exists a $\lambda$-calculus term $M$ where $\Gamma' \vdash M : A$, where $\Gamma' = \{x_i : \phi_i \mid \phi_i \in \Gamma\}$.*

This gives a notion of reducing a proof, since the type judgements of the $\lambda$-calculus are preserved through reduction.

**Example.** The formula $A \to A$ is valid in intuitionistic logic. Consider the following natural deduction proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\text{Ax})}{A \vdash A \to A}\,(\to \text{I}) \qquad \dfrac{}{A \vdash A}\,(\text{Ax})}{A \vdash A}\,(\to \text{E})}{\emptyset \vdash A \to A}\,(\to \text{I})$$

This proof has some redundencies - the ($\to$ E) is unnecessary, from the axiom $A \vdash A$ you can reach the desired result immediately from ($\to$ I).

This proof corresponds to the $\lambda$-term $\lambda x.(\lambda y.y)x$, substituting (Ax) for variables, ($\to$ I) for $\lambda$ abstractions and ($\to$ E) for application. But $\lambda x.(\lambda y.y)x \rightsquigarrow \lambda x.x$, which is in normal form. The corresponding natural deduction proof for $\lambda x.x$ is:

$$\dfrac{\dfrac{}{A \vdash A}\,(\text{Ax})}{\emptyset \vdash A \to A}\,(\to \text{I})$$

substituting (Ax) for $x$ and ($\to$ I) for $\lambda x$. This is the exact proof above, but with the redundencies removed. So we can normalise natural deduction proofs.

A followup question might be if we could re-create this correspondence for classical logic, or sequent calculus-style deductions. The $\lambda\mu$-calculus (Chapter 5) has a similar correspondence with natural deduction-style classical logic, and the $\bar{\lambda}\mu\tilde{\mu}$-calculus (Chapter 6) has a correspondence with sequent calculus-style classical logic.

# Chapter 4

# Call-by-name and call-by-value strategies

Reduction strategies exist to provide deterministic ways of reducing terms. For example, for every $\lambda$-calculus term $M$, if $M \rightsquigarrow_n N$, i.e. $M$ reduces in a single step to $N$ via the call-by-name reduction strategy, then $N$ is the unique such term to satisfy $M \rightsquigarrow_n N$. In other words, $\rightsquigarrow_n$ is a (partial) function from terms to terms. The same is true for $\rightsquigarrow_v$.

The general idea of the call-by-name strategy is to immediately apply the argument to a $\lambda$ term and not reduce it until it is needed. Call-by-value does the opposite: the argument to a $\lambda$ term is reduced until it is not of the form $st$ any more (so the subterm will never reduce on its own again) and is then applied.

To describe what reductions can take place under different circumstances, we will use contexts.

## 4.1 Contexts

Contexts and reduction frames are useful tools to formalise reduction strategies. They restrict how reduction propagates through subterms.

As an example, in the $\lambda$-calculus (see Chapter 3), reduction can take place in any subterm of any term. However, in the call-by-name strategy, reduction can only propagate through the left-hand side of an application and in no other subterm/term.

**Definition 4.1** (Contexts)**.** A context of a calculus is a term with a hole which can be filled by a term or another context. We denote it by $E$ or $E[\,]$, where $[\,]$ is the hole.

Let $N$ be a term. $E[N]$ is a term formed by replacing the hole in the context $E[\,]$ by the term $N$. Note that the hole may be several subterms deep in the context.

$E_1[E_2[\,]]$ is a context formed by replacing the hole in the context $E_1[\,]$ by another context $E_2[\,]$, such that the resulting context has the hole from $E_2$ nested inside $E_1$.

**Example.** In the lambda calculus, $y(\lambda x.[\,])$ is a context. Name it $E_1[\,]$.

$E_1[xx]$ is the term $y(\lambda x.xx)$.

Let $E_2[\,]$ be the context $[\,]z$. $E_1[E_2[\,]]$ is the context $y(\lambda x.[\,]z)$.

These contexts can be used to restrict reduction using a set of them called reduction frames.

**Definition 4.2** (Reduction frames)**.** A set of reduction frames is a set of contexts.

**Example.** Define a set of reduction frames of the $\lambda$-calculus as follows:

$$E ::= [\,] \mid \lambda x.E \mid Et$$

Restrict reduction in the $\lambda$-calculus to the following axioms:

$$(\lambda x.s)t \rightsquigarrow s\,[t/x]$$

$$\frac{s \to s'}{E[s] \rightsquigarrow E[s']}$$

This only allows reduction to propagate underneath a $\lambda$ abstraction or on the left-hand-side of an application. The reduction

$$x((\lambda y.y)z) \rightsquigarrow xz$$

is not valid in this reduction system, as the subterm reduced is on the right-hand-side of an application.

## 4.2 Call-by-name

In call-by-name $\lambda$-calculus, other than primitive beta reduction, only the left-hand-side of an application can be reduced. So sub-terms are only reduced when something needs to be applied to them.

**Definition 4.3** (Call-by-name $\lambda$-calculus syntax)**.** The syntax of the call-by-name subset of the lambda calculus is defined as:

$$s, t ::= x \mid \lambda x.s \mid st$$

The reduction frames of the call-by-name subset of the lambda calculus are defined as:

$$E ::= [\,] \mid Et$$

**Definition 4.4** (Call-by-name $\lambda$-calculus reduction)**.** The reduction relation $\rightsquigarrow_n$ is defined as:

$$(\lambda x.s)t \rightsquigarrow_n s\,[t/x]$$

$$\frac{s \rightsquigarrow_n s'}{E[s] \rightsquigarrow_n E[s']}$$

**Example.** Consider the term $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$.

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \rightsquigarrow_n \lambda y.y$$

## 4.3 Call-by-value

First, we need to define the syntax and reduction rules of the call-by-value subset of the lambda calculus. In call-by-value, only arguments which cannot ever possibly reduce on their own are applied. These form a subset of terms called values, and only consist of variables and $\lambda$ abstractions. Only once an arguemnt is reduced to a value can it ever be applied to a function.

**Definition 4.5** (Call-by-value $\lambda$-calculus syntax)**.** The syntax of the call-by-value subset of the lambda calculus is defined as:

$$u, v ::= x \mid \lambda x.s$$
$$s, t ::= v \mid st$$

The reduction frames of the call-by-value subset of the lambda calculus are defined as:

$$E ::= [\,] \mid Et \mid vE$$

**Definition 4.6** (Call-by value $\lambda$-calculus reduction)**.** The reduction relation $\rightsquigarrow_v$ from terms to terms is defined as:

$$(\lambda x.s)v \rightsquigarrow_v s\,[v/x]$$

$$\frac{s \rightsquigarrow_v s'}{E[s] \rightsquigarrow_v E[s']}$$

**Example.** Consider the term $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$.

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \rightsquigarrow_v (\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$$
$$\rightsquigarrow_v \ldots$$

# Chapter 5

# $\lambda\mu$ calculus

There have been many calculi invented to algorithmically model the proof systems in Chapter 2, based on the discovery of the simply typed $\lambda$-calculus' isomorphism to natural deduction-style intuitionistic logic proofs 3.3.

One of the first was by Parigot presented the calculus $\lambda\mu$ in 1992 to model (a subset of) natural deduction-style proofs in classical logic [4]. From this paper's perspective, this calculus is a stepping stone to model sequent calculus-style proofs in classical logic, and can give insight into how the $\mu$ construct is necessary for modelling classical logic in natural deduction.

## 5.1 Syntax and reduction

**Definition 5.1** ($\lambda\mu$ terms [4])**.** The $\lambda\mu$-calculus terms are defined as follows:

$$s, t ::= x \mid \lambda x.s \mid st \mid \mu\alpha.\,[\beta]\,s$$

$\lambda\mu$ is an extension of the $\lambda$-calculus, in that terms only including $x$, $\lambda x.s$ and $st$ constructs behave exactly the same their corresponding $\lambda$-calculus terms. The construct $[\beta]\,s$ can be thought of as a term $s$ *labelled* with $\beta$. The construct $\mu\alpha.\,[\beta]\,s$ is another kind of function like $\lambda x.s$; instead of subsituting its argument once and then continuing with the reduction of its immediate subterm, it re-directs its argument to its subterms labelled by $\alpha$ (which there may be many of). In this way the term $\mu\alpha.\,[\beta]\,s$ can be modelled by the $\lambda$-calculus term $\lambda x.sy$, replacing $\alpha$ with $x$ and $\beta$ with $y$.

But unlike the $\lambda$ abstraction, the $\mu$ abstraction doesn't disappear afterwards - it can continually feed arguments to its corresponding labelled subterms a potentially infinite amount of times. So it can be thought of as a way to model functions with an unknown number of arguments.

To represent this intuition formally, we need a different kind of substitution for the greek variables (the labelling variables).

**Definition 5.2** ($\lambda\mu$ substitution)**.** Terms are substituted for latin variables in the usual way. The

construct $t \cdot \gamma$ is substituted for greek variables in the following way:

$$\begin{aligned}
x\,[t\cdot\gamma/\alpha] &= x \\
(\lambda x.s)\,[t\cdot\gamma/\alpha] &= \lambda x.s\,[t\cdot\gamma/\alpha] \\
(s_1 s_2)\,[t\cdot\gamma/\alpha] &= s_1\,[t\cdot\gamma/\alpha]\; s_2\,[t\cdot\gamma/\alpha] \\
(\mu\beta.\,[\alpha]\,s)\,[t\cdot\gamma/\alpha] &= \mu\beta.\,[\gamma]\,(s\,[t\cdot\gamma/\alpha])t \\
(\mu\beta.\,[\delta]\,s)\,[t\cdot\gamma/\alpha] &= \mu\beta.\,[\delta]\,s\,[t\cdot\gamma/\alpha] \qquad \delta \neq \alpha
\end{aligned}$$

Now the reduction rules.

**Definition 5.3** ($\lambda\mu$ reduction rules)**.** $\lambda\mu$ reduction rules are defined as follows:

$$\begin{aligned}
(\lambda x.s)t &\rightsquigarrow_\beta s\,[t/x] \\
(\mu\alpha.\,[\beta]\,s)t &\rightsquigarrow_{(\mu)} \mu\gamma.\,[\beta]\,(s\,[t\cdot\gamma/\alpha])
\end{aligned}$$

## 5.2 Simple type system

The typing rules for the borrowed terms from the $\lambda$-calculus are similar to their simple types in the $\lambda$-calculus (see Section 3.2). Recall that a typing judgement in the $\lambda$-calculus is of the form $\Gamma \vdash s : A$, where $\Gamma$ is a partial function from (latin) variables to types. Via the Curry-Howard isomorphism, $\Gamma$ represents the current assumptions. These assumptions are all positive - i.e. have no negations in them.

But in $\lambda\mu$ we also have greek variables. We can think of the greek variables representing negative assumptions - i.e. the negation of a positive formula. We keep a separate record for their types - $\Sigma$. We use the judgement $\Gamma \vdash s : A \mid \Sigma$ to mean, via a Curry-Howard style correspondence, $\Gamma \cup \neg\Sigma \vdash A$, where $\neg\Sigma$ is the negation of all formulae in $\Sigma$. Notice that in classical sequent calculus, from $\Gamma \vdash A, \Sigma$ you can deduce $\Gamma \cup \neg\Sigma \vdash A$ and vice versa.

*Notation.* We call $(x, y, z)$ term variables, and $(\alpha, \beta, \gamma)$ context variables.

**Definition 5.4** (Simple $\lambda\mu$ type assignments)**.** We take $\Gamma$ to be a partial function of term variables to types $(A, B, C)$ and $\Sigma$ a partial function of context variables to types.

An $\lambda\mu$ term $s$ is typed in the following typing judgement: $\Gamma \vdash s : A \mid \Sigma$.

The type assignments are:

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Sigma}\ (\text{Ax})$$

$$\frac{\Gamma, x : A \vdash s : B \mid \Sigma}{\Gamma \vdash \lambda x.s : A \to B \mid \Sigma}\ (\to \text{I})$$

$$\frac{\Gamma \vdash s : A \to B \mid \Sigma \qquad \Gamma \vdash t : A \mid \Sigma}{\Gamma \vdash st : B \mid \Sigma}\ (\to \text{E})$$

$$\frac{\Gamma \vdash s : A \mid \alpha : A, \Sigma}{\Gamma \vdash \mu\alpha.\,[\alpha]\,s : A \mid \Sigma}\ (\mu)$$

$$\frac{\Gamma \vdash s : B \mid \alpha : A, \beta : B, \Sigma}{\Gamma \vdash \mu\alpha.\, [\beta]\, s : A \mid \beta : B, \Sigma}\ (\mu)$$

The $(\mu)$ inference rule represents proof by contradiction. To prove $A$, you assume $\neg A$, and you reach $\bot$ - you prove $B$ even though you have assumed $\neg B$. So $\Gamma \vdash \mu\alpha.\, [\beta]\, s : A \mid \beta : B, \Sigma$ let's $\alpha$ witness $\neg A$, and then needs that $M : B$ even though you have a witness for $\neg B$ - $\beta$.

In fact, if you separate out the term $\mu\alpha.\, [\beta]\, s$ into two constructs: $\mu\alpha.s$ and $[\beta]\, s$, then $[\beta]\, s : \bot$ [12]. This calculus is called $\Lambda\mu$. You can directly model proof by contradiction via the inference rule:

$$\frac{\Gamma \vdash s : \bot \mid \alpha : A, \Sigma}{\Gamma \vdash \mu\alpha.s : A \mid \Sigma}$$

# Chapter 6

# $\bar{\lambda}\mu\tilde{\mu}$ calculus

Parigot developed the $\lambda\mu$ calculus (see Chapter 5) to model a subset of natural deduction-style classical logic proofs. These proofs features a separate kind of variable store (context variables) to keep track of negative assumptions.

But since sequent calculus enjoys better proof-theory properties, such as the sub-formula property, (see Section 2.3), a similar calculus to $\lambda\mu$ that models sequent calculus proofs with focus would be ideal.

In 2000, Curien and Herbelin presented the $\bar{\lambda}\mu\tilde{\mu}$-calculus to model a subset of sequent calculus-style proofs for classical logic [1].[1] It is based on the call-by-name and call-by-value strategies of the $\lambda$-calculus, and encodes them both into its reductions. See Section 8.2 for how $\bar{\lambda}\mu\tilde{\mu}$'s syntax and reductions can be constructed from call-by-name and call-by-value $\lambda$-calculus strategies.

Instead of modelling the sequent calculus inference rules directly with terms, it models sequent calculus proofs with focus. In a sequent a single formula may be focused on, either in the antecedent or the succedent, or no formula may be foucsed on. $\bar{\lambda}\mu\tilde{\mu}$ has three different constructs to represent these three different types of sequencts.

$\bar{\lambda}\mu\tilde{\mu}$'s simple reduction rules and easy restrictions for call by name/call by value strategies are ideal for working with computationally; however there is no easy way to model cut reductions for sequent calculus proofs. In fact, not all cut reductions can be modelled.

## 6.1 Proofs with focus in sequent calculus

A sequent with focus acts exactly the same as a normal sequent when it interacts with the rules of sequent calculus. However, it makes syntactic changes to the rules: only the active formula (the formula being focused on) can be changed/manipulated.

**Definition 6.1** (Sequents with focus)**.** A sequent with focus is a sequent $\Gamma \vdash \Sigma$, where one formula in either $\Gamma$ or $\Sigma$ is being focused on:

$$\Gamma \vdash A \mid \Sigma$$

$$\Gamma \mid A \vdash \Sigma$$

[1] This paper could have chosen to work with the $\mathcal{X}$-calculus [6], which also models sequent calculus-style classical logic proofs, instead of $\bar{\lambda}\mu\tilde{\mu}$. The decision to work with the $\bar{\lambda}\mu\tilde{\mu}$-calculus was because it can be viewed as an abstract machine in which to run the $\lambda$-calculus (see Chapter 8), and so would more naturally model Call by Push Value (see Chapter 7).

where $A$ is the formula being focused on. We call $A$ the active formula.

We can think of $\Gamma$ or $\Sigma$ being unordered, except for the focused formula, which is always the closest to the $\vdash$ sign. Some of the rules change to represent this focus on an active formula.

**Definition 6.2** (Sequnt calculus with focus)**.** Most structural rules are the same, with the restriction to only involving non-active formulae. (Cut) changes as so:

$$\frac{\Gamma \vdash A \mid \Sigma \qquad \Delta \mid A \vdash \Lambda}{\Gamma, \Delta \vdash \Sigma, \Lambda} \text{(Cut)}$$

The operational formulae are altered as so:

$$\frac{}{\Gamma \mid A \vdash A, \Sigma} \text{(Ax-A)}$$

$$\frac{}{\Gamma, A \vdash A \mid \Sigma} \text{(Ax-R)}$$

$$\frac{A, \Gamma \vdash B \mid \Sigma}{\Gamma \vdash A \to B \mid \Sigma} (\to IS)$$

$$\frac{\Gamma \vdash A \mid \Sigma \qquad \Delta \mid B \vdash \Lambda}{\Gamma, \Delta \mid A \to B \vdash \Sigma, \Lambda} (\to IA)$$

Some of the drawbacks of using proofs with focus, is that you need extra manipulation to always be focusing on the formula you want to use in a particular rule.

## 6.2 $\bar{\lambda}\mu\tilde{\mu}$ syntax and reduction

Each term in the $\bar{\lambda}\mu\tilde{\mu}$-calculus belongs to one of three categories: whether the final sequent in the proof it represents has its active formula in the antecedent, the succedent or there is no active formula (i.e. a cut has just occurred).

**Definition 6.3** ($\bar{\lambda}\mu\tilde{\mu}$ terms)**.** $\bar{\lambda}\mu\tilde{\mu}$-calculus terms are split into three categories: commands ($c$), values ($v$), and contexts ($e$). They are defined as follows:

$$\begin{aligned} c &::= \langle v \mid e \rangle \\ v &::= x \mid \mu\alpha.c \mid \lambda x.v \\ e &::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e \end{aligned}$$

As with the $\lambda\mu$-calculus (see Chapter 5), $x$ is bound in the term $\lambda x.v$, and $\alpha$ is bound in the term $\mu\alpha.c$. We now say that $x$ is bound in the term $\tilde{\mu}x.c$.

**Definition 6.4** ($\bar{\lambda}\mu\tilde{\mu}$ reduction rules)**.** $\bar{\lambda}\mu\tilde{\mu}$-calculus reduction rules are split into two categories: logical and extensional.

The logical rules are defined as follows:

$$\langle \lambda x.v_1 \mid v_2 \cdot e \rangle \rightsquigarrow_{\beta} \langle v_2 \mid \tilde{\mu}x.\langle v_2 \mid e \rangle \rangle$$
$$\langle \mu\alpha.c \mid e \rangle \rightsquigarrow_{(\mu)} c\,[e/\alpha]$$
$$\langle v \mid \tilde{\mu}x.c \rangle \rightsquigarrow_{(\tilde{\mu})} c\,[v/x]$$

The extensional rules are defined as follows:

$$\lambda x.\mu\alpha.\langle v \mid x \cdot \alpha \rangle \rightsquigarrow_{(\eta)} v \qquad x, \alpha \notin fv(v)$$
$$\mu\alpha.\langle v \mid \alpha \rangle \rightsquigarrow_{(\eta\mu)} v \qquad \alpha \notin fv(v)$$
$$\tilde{\mu}x.\langle x \mid e \rangle \rightsquigarrow_{(\eta\tilde{\mu})} e \qquad x \notin fv(e)$$

As with the $\lambda$-calculus (see Chapter 3), reductions can occur in any subterm, a redex is a term that can be reduced, and a term in normal form is a term that cannot be reduced.

This reduction is not confluent - consider the following example:

**Example.**

$$\langle \mu\alpha.\langle y \mid \beta \rangle \mid \tilde{\mu}x.\langle z \mid \gamma \rangle \rangle \rightsquigarrow_{(\mu)} \langle y \mid \beta \rangle$$
$$\langle \mu\alpha.\langle y \mid \beta \rangle \mid \tilde{\mu}x.\langle z \mid \gamma \rangle \rangle \rightsquigarrow_{(\tilde{\mu})} \langle z \mid \gamma \rangle$$

To combat this lack of confluence, two reduction limitations were introduced: call-by-name and call-by-value. Call by name reduction gives priority to $\rightsquigarrow_{(\tilde{\mu})}$ - i.e. $\rightsquigarrow_{(\mu)}$ can only be executed if the context is not of the form $\tilde{\mu}x.c$. Call by value reduction gives priority to $\rightsquigarrow_{(\mu)}$ - i.e. $\rightsquigarrow_{(\tilde{\mu})}$ can only be executed if the value is not of the form $\mu\alpha.c$.

**Definition 6.5** (Call by name reduction limitations)**.** Refine the $\bar{\lambda}\mu\tilde{\mu}$ syntax to:

$$v ::= x \mid \mu\alpha.c \mid \bar{\lambda}x.v$$
$$c ::= \langle v \mid e \rangle$$
$$e ::= E \mid \tilde{\mu}x.c$$
$$E ::= \alpha \mid v \cdot e$$

and restrict the reduction relation $\rightsquigarrow_{(\mu)}$:

$$\langle \mu\alpha.c \mid E \rangle \rightsquigarrow_{(\mu)} c\,[E/\alpha]$$

**Definition 6.6** (Call by value reduction limitations)**.** Refine the $\bar{\lambda}\mu\tilde{\mu}$ syntax to:

$$V ::= x \mid \bar{\lambda}x.v$$
$$v ::= V \mid \mu\alpha.c$$
$$c ::= \langle v \mid e \rangle$$
$$e ::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e$$

and restrict the reduction relation $\rightsquigarrow_{(\tilde{\mu})}$:

$$\langle V \mid \tilde{\mu}x.c \rangle \rightsquigarrow_{(\tilde{\mu})} c\,[V/x]$$

## 6.3 Simple type system

**Definition 6.7** (Simple $\bar{\lambda}\mu\tilde{\mu}$ type assignments)**.** We take $\Gamma$ to be a partial function of value variables $(x, y, z)$ to types $(A, B, C)$, and $\Sigma$ to be a partial function of context variables $(\alpha, \beta, \gamma)$ to types.

There are three different styles of typing judgements for each different category of term.

Commands are typed in the format $c : (\Gamma \vdash \Sigma)$. Their type assignment is:

$$\frac{\Gamma \vdash v : A \mid \Sigma \qquad \Gamma \mid e : A \vdash \Sigma}{\langle v \mid e \rangle : (\Gamma \vdash \Sigma)} \text{ (command)}$$

Values are typed in the format $\Gamma \vdash v : A \mid \Sigma$. Their type assignments are:

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Sigma} \text{ (Ax-value)}$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Sigma)}{\Gamma \vdash \mu\alpha.c : A \mid \Sigma} \text{ ($\mu$-value)}$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Sigma}{\Gamma \vdash \lambda x.v : A \to B \mid \Sigma} \text{ ($\lambda$-value)}$$

Contexts are typed in the format $\Gamma \mid e : A \vdash \sigma$. Their type assignments are:

$$\frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Sigma} \text{ (Ax-context)}$$

$$\frac{c : (\Gamma, x : A \vdash \Sigma)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Sigma} \text{ ($\tilde{\mu}$-context)}$$

$$\frac{\Gamma \vdash v : A \mid \Sigma \qquad \Gamma \mid e : B \vdash \Sigma}{\Gamma \mid v \cdot e : A \to B \vdash \Sigma} \text{ ($\cdot$-context)}$$

In sequent calculus with focus (see Section 6.1), we can view each $\bar{\lambda}\mu\tilde{\mu}$ values as focuses on formulae in the succedent, contexts as focuses in the antecedent and commands as proofs with no final focus. In particular, $x$ models (Ax-R), $\bar{\lambda}x.v$ models ($\to$ IS), $\alpha$ models (Ax-L), $v \cdot e$ models ($\to$ IA) and $\langle v \mid e \rangle$ models (Cut). The interesting cases are $\mu\alpha$ and $\tilde{\mu}x$. $\mu\alpha$ takes a sequent with no focus and chooses a formula in the succedent to focus on (the one labelled $\alpha$). $\tilde{\mu}x$ takes a sequent with no focus and chooses a formula in the antecedent to focus on (the one labelled $x$).

From now on, we consider traditional sequent calculus (i.e. all sequents have no focus).

Below is an example highlighting the way this typing system corresponds to formulae and proofs in sequent calculus-style classical logic.

**Example.** Pierce's law $((A \to B) \to A) \vdash A$ can be proved by the term
$\langle x \mid \lambda y.\mu\gamma.\, \langle y \mid \alpha \rangle \cdot \alpha \rangle$

Let $\Gamma = x : (A \to B) \to A$.

Let $\Sigma = \alpha : A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : A \vdash y : A \mid \Sigma, \gamma : B} \quad \dfrac{}{\Gamma, y : A \mid \alpha : A \vdash \Sigma, \gamma : B}}{\langle y \mid \alpha \rangle : (\Gamma, y : A \vdash \Sigma, \gamma : B)}}{\Gamma, y : A \vdash \mu\gamma.\langle y \mid \alpha \rangle : B \mid \Sigma}}{\Gamma \vdash \lambda y.\mu\gamma.\langle y \mid \alpha \rangle : A \to B \mid \Sigma} \quad \dfrac{}{\Gamma \mid \alpha : A \vdash \Sigma}}{\dfrac{\Gamma \vdash x : (A \to B) \to A \mid \Sigma \qquad \Gamma \mid \lambda y.\mu\gamma.\langle y \mid \alpha \rangle \cdot \alpha : (A \to B) \to A \vdash \Sigma}{\langle x \mid \lambda y.\mu\gamma.\langle y \mid \alpha \rangle \cdot \alpha \rangle : (\Gamma \vdash \Sigma)}}$$

The following terms model the implicational sequent calculus rules.

**Lemma 6.8.** *The rule (Ax) from sequent calculus can be modelled as follows:*

$$\dfrac{}{\langle x \mid \alpha \rangle : (\Gamma, x : A \vdash \Sigma, \alpha : A)}$$

*The rule ($\to$ IS) can be modelled as follows:*

$$\dfrac{c : (\Gamma, x : A \vdash \Sigma, \alpha : B)}{\langle \lambda x.\mu\alpha.c \mid \beta \rangle : (\Gamma \vdash \Sigma, \beta : A \to B)}$$

*The rule ($\to$ IA) can be modelled as follows:*

$$\dfrac{c_1 : (\Gamma \vdash \Sigma, \alpha : A) \qquad c_2 : (\Gamma, x : B \vdash \Sigma)}{\langle y \mid \mu\alpha.c_1 \cdot \tilde{\mu}x.c_2 \rangle : (\Gamma, y : A \to B \vdash \Sigma)}$$

*The rule (Cut) can be modelled as follows:*

$$\dfrac{c_1 : (\Gamma \vdash \Sigma, \alpha : A) \qquad c_2 : (\Gamma, x : A \vdash \Sigma)}{\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle : (\Gamma \vdash \Sigma)}$$

However, using this we can see that reduction in the $\bar{\lambda}\mu\tilde{\mu}$-calculus doesn't model every cut reduction in proofs.

**Example.** Example from [6], page 37. The following sequent calculus proof (1) can be cut-reduced in two ways:

$$\dfrac{\dfrac{\Gamma, A \vdash B, \Sigma}{\Gamma \vdash A \to B, \Sigma} \quad \dfrac{\Gamma \vdash A, \Sigma \qquad \Gamma, B \vdash \Sigma}{\Gamma, A \to B \vdash \Sigma}}{\Gamma \vdash \Sigma}$$

First reduction (2):

$$\dfrac{\Gamma \vdash A, \Sigma \qquad \dfrac{\Gamma, A \vdash B, \Sigma \qquad \dfrac{\Gamma, B \vdash \Sigma}{\Gamma, A, B \vdash \Sigma}}{\Gamma, A \vdash \Sigma}}{\Gamma \vdash \Sigma}$$

Second reduction (3):

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A, \Sigma}{\Gamma \vdash A, B, \Sigma} \qquad \Gamma, A \vdash B, \Sigma}{\Gamma \vdash B, \Sigma} \qquad \Gamma, B \vdash \Sigma}{\Gamma \vdash \Sigma}$$

Say $c_1 : (\Gamma, x : A \vdash \alpha : B, \Sigma)$, $c_2 : (\Gamma \vdash \gamma : A, \Sigma)$, $c_3 : (\Gamma, z : B \vdash \Sigma)$.

Then we have that

- The proof (1) has the corresponding $\bar{\lambda}\mu\tilde{\mu}$ term
$$C1 = \langle \lambda x.v_1 \mid v_2 \cdot e \rangle$$
- The proof (2) has the corresponding $\bar{\lambda}\mu\tilde{\mu}$ term
$$C2 = \langle v_2 \mid \tilde{\mu}x.\, \langle v_1 \mid e \rangle \rangle$$
- The proof (3) has the corresponding $\bar{\lambda}\mu\tilde{\mu}$ term
$$C3 = \langle \mu\alpha.\, \langle v_2 \mid \tilde{\mu}x.\, \langle v_1 \mid \alpha \rangle \rangle \mid e \rangle$$

However, $C1 \rightsquigarrow^* C2$ but we don't have that $C1 \rightsquigarrow^* C3$.

## 6.4 Interpreting the $\lambda$ calculus

A translation of $\lambda$ into $\bar{\lambda}\mu\tilde{\mu}$ does several things: it shows how computation (function application) is run in $\bar{\lambda}\mu\tilde{\mu}$, it gives justification to how call by name and call by value reduction should be defined, and therefore gives a starting point for interpreting and extending Call By Push Value (Chapter 7) in $\bar{\lambda}\mu\tilde{\mu}$.

*Recall.* The terms of the $\lambda$-calculus are defined as

$$s, t ::= x \mid \lambda x.s \mid st$$

*Recall.* The $\beta$-reduction rule is defined as

$$(\lambda x.s)t \rightsquigarrow_\beta s\,[t/x]$$

In $\bar{\lambda}\mu\tilde{\mu}$, the values $x$ and $\bar{\lambda}x.v$ are natural allegories of $x$ and $\lambda x.s$, not only in how they look but how they behave - a $\bar{\lambda}\mu\tilde{\mu}$ value $x$ is a value to be substituted, and $\bar{\lambda}x.v$ is a function to be fed an argument, and eventually reduced.

So the difficult part of the interpretation is $st$. It must be a $\bar{\lambda}\mu\tilde{\mu}$ value, since the other interpretations have been and there is nothing to differentitate this term from another. It must also have a $\bar{\lambda}\mu\tilde{\mu}$ command somewhere to be able to reduce, since the term $st$ can possibly reduce.

One way to validate these conditions is to intepret $st$ to a $\bar{\lambda}\mu\tilde{\mu}$ value of the form $\mu\alpha.\, \langle v \mid e \rangle$. We can view the term $st$ as feeding the argument $t$ to $s$ - something which $\bar{\lambda}\mu\tilde{\mu}$ commands do if the right-hand-side of the command is a stack (i.e. of the form $v \cdot e$).

Using this intuition, we can define the interepretation of the $\lambda$-calculus.

**Definition 6.9** (Intepretation of $\lambda$-calculus)**.** The interpretation operator

$$[\![\cdot]\!]^{\lambda} : \lambda \text{ term} \to \bar{\lambda}\mu\tilde{\mu} \text{ values}$$

is defined as:

$$\begin{aligned} [\![x]\!]^{\lambda} &= x \\ [\![\lambda x.s]\!]^{\lambda} &= \bar{\lambda}x.[\![s]\!]^{\lambda} \\ [\![st]\!]^{\lambda} &= \mu\alpha.\left\langle [\![s]\!]^{\lambda} \mid [\![t]\!]^{\lambda} \cdot \alpha \right\rangle \end{aligned}$$

This preserves beta reduction in the $\lambda$-calculus.

**Lemma 6.10** (Beta-reduction preservation)**.** *Let $s, t$ be $\lambda$-calculus terms. If $s \rightsquigarrow^* t$, then $[\![s]\!]^{\lambda} \rightsquigarrow^* [\![t]\!]^{\lambda}$.*

*Proof.* The interesting case is primitive beta reduction, i.e. that $[\![(\lambda x.s)t]\!]^{\lambda} \rightsquigarrow^* [\![s]\!]^{\lambda}\,[{}^{[\![t]\!]^{\lambda}}/_{x}]$.

$$\begin{aligned} [\![(\lambda x.s)t]\!]^{\lambda} &= \mu\alpha.\left\langle \bar{\lambda}x.[\![s]\!]^{\lambda} \mid [\![t]\!]^{\lambda} \cdot \alpha \right\rangle \\ &\rightsquigarrow_{\beta} \mu\alpha.\left\langle [\![t]\!]^{\lambda} \mid \tilde{\mu}x.\left\langle [\![s]\!]^{\lambda} \mid \alpha \right\rangle \right\rangle \\ &\rightsquigarrow_{(\tilde{\mu})} \mu\alpha.\left\langle [\![s]\!]^{\lambda}\,[{}^{[\![t]\!]^{\lambda}}/_{x}] \mid \alpha \right\rangle \\ &\rightsquigarrow_{(\eta\mu)} [\![s]\!]^{\lambda}\,[{}^{[\![t]\!]^{\lambda}}/_{x}] \end{aligned}$$

□

We can preserve call-by-name and call-by-value reduction in the $\lambda$-calculus (see Chapter **??**) via the respective reduction limitations in Definitions 6.5 6.6.

**Example.** Consider $s = (\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$.

$$\begin{aligned} s &\rightsquigarrow_{n} \lambda y.y \\ s &\rightsquigarrow_{v} s \rightsquigarrow_{v} s \rightsquigarrow_{v} \ldots \end{aligned}$$

Via the $\bar{\lambda}\mu\tilde{\mu}$ call-by-name strategy, i.e. in a situation of $\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle$ you always reduce $\tilde{\mu}$, we can reduce:

$$\begin{aligned} [\![s]\!]^{\lambda} &= \mu\alpha.\left\langle \bar{\lambda}xy.y \mid (\mu\beta.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle \mid (\bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \cdot \beta \right\rangle) \cdot \alpha \right\rangle \\ &\rightsquigarrow_{\beta} \mu\alpha.\left\langle \mu\beta.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle \mid (\bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \cdot \beta \right\rangle \mid \tilde{\mu}x.\left\langle \bar{\lambda}y.y \mid \alpha \right\rangle \right\rangle \\ &\rightsquigarrow_{(\tilde{\mu})} \mu\alpha.\left\langle \bar{\lambda}y.y \mid \alpha \right\rangle \\ &\rightsquigarrow_{(\eta\mu)} \bar{\lambda}y.y \\ &= [\![\lambda y.y]\!]^{\lambda} \end{aligned}$$

Via the $\bar{\lambda}\mu\tilde{\mu}$ call-by-value strategy, i.e. in a situation of $\langle \mu\alpha.c_1 \mid \tilde{\mu}x.c_2 \rangle$ you always reduce $\mu$, we can reduce:

$$\begin{aligned} [\![s]\!]^{\lambda} &= \mu\alpha.\left\langle \bar{\lambda}xy.y \mid (\mu\beta.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle \mid (\bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \cdot \beta \right\rangle) \cdot \alpha \right\rangle \\ &\rightsquigarrow_{\beta} \mu\alpha.\left\langle \bar{\lambda}xy.y \mid (\mu\beta.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \mid \tilde{\mu}x.\left\langle \mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle \mid \beta \right\rangle \right\rangle) \cdot \alpha \right\rangle \\ &\rightsquigarrow_{(\tilde{\mu})} \mu\alpha.\left\langle \bar{\lambda}xy.y \mid (\mu\beta.\left\langle \mu\gamma.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \mid (\bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle)) \cdot \gamma \right\rangle \mid \beta \right\rangle) \cdot \alpha \right\rangle \\ &\rightsquigarrow_{(\mu)} \mu\alpha.\left\langle \bar{\lambda}xy.y \mid (\mu\beta.\left\langle \bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle \mid (\bar{\lambda}x.\mu\gamma.\left\langle x \mid x \cdot \gamma \right\rangle) \cdot \beta \right\rangle) \cdot \alpha \right\rangle \\ &= [\![s]\!]^{\lambda} \end{aligned}$$

We would also like this translation to justify the backwards simulation of call-by-name and call-by-value in $\bar{\lambda}\mu\tilde{\mu}$. However, those limitations do not enforce enough restrictions on the reduction of terms.

**Example.** Consider the $\lambda$-calculus term $(\lambda x.(\lambda y.y)x)z$. Under both call-by-name and call-by-value, this should reduce in one step to $(\lambda y.y)z$ - in particular, no reduction underneath the $\lambda$ abstraction should occur. However,

$$\begin{aligned}
[\![(\lambda x.(\lambda y.y)x)z]\!]^{\lambda} &= \mu\alpha.\left\langle \bar{\lambda}x.\mu\beta.\left\langle \bar{\lambda}y.y \mid x \cdot \beta \right\rangle \mid z \cdot \alpha \right\rangle \\
&\rightsquigarrow_{\beta} \mu\alpha.\left\langle \bar{\lambda}x.\mu\beta.\langle x \mid \tilde{\mu}y.\langle y \mid \beta\rangle\rangle \mid z \cdot \alpha \right\rangle \\
&\rightsquigarrow_{(\tilde{\mu})} \mu\alpha.\left\langle \bar{\lambda}x.\mu\beta.\langle x \mid \beta\rangle \mid z \cdot \alpha \right\rangle \\
&\rightsquigarrow_{(\eta\mu)} \mu\alpha.\left\langle \bar{\lambda}x.x \mid z \cdot \alpha \right\rangle \\
&= [\![(\lambda x.x)z]\!]^{\lambda}
\end{aligned}$$

None of these reductions are illegal by either the call-by-name or call-by-value limitations, but the reduction $(\lambda x.(\lambda y.y)x)z \rightsquigarrow (\lambda x.x)z$ is not a single step reduction for either call-by-name or call-by-value $\lambda$-calculus.

Assignable types are also preserved by the interpretation (see Section 3.2) Since $\lambda$ terms are always mapped into $\bar{\lambda}\mu\tilde{\mu}$ values, we type them as so - that is $\Gamma \vdash v : A \mid \Sigma$.

**Lemma 6.11** (Type preservation)**.** *Let $s$ be a $\lambda$-calculus term. If $\Gamma \vdash s : A$, then $\Gamma \vdash [\![s]\!]^{\lambda} : A \mid \Sigma$ for any set of typed context variables $\Sigma$.*

# Chapter 7

# Call by Push Value

Call by Push Value (CBPV), presented by Levy, is a calculus designed to make the execution order of a lambds-based term explicit [2]. There is a deterministic way to reduce every term a single step, if a term is reducible.

In intuitionistic logic-based calculi with different reduction strategies (for example, call by name in the lambda calculus), if you want to specify operational semantics, or denotational semantics, or effects, then you need to provide each semantics for every reduction strategy. Instead, CBPV allows you to only create one interpretation from your base calculus into CBPV for each reduction strategy, allowing you to gain all these other specifications for free.

In this paper, we use the syntax from [13] and ignore sums and products[1].

## 7.1 Syntax and reduction

**Definition 7.1** (CBPV terms)**.** There are two categories of CBPV-terms: values $(V, W)$ and computations $(M, N)$.

They are defined as follows:

$$\begin{aligned} \textit{(values)}\ V, W &::= x \mid \{M\} \\ \textit{(computations)}\ M, N &::= \lambda x.M \mid MV \mid V! \mid \texttt{return}\ V \mid \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \end{aligned}$$

Read $\{M\}$ as "thunk" $M$, $V!$ as "force" $V$ and $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$ as let $x$ be $M$ in $N$.

We say that $x$ is bound in the terms $\lambda x.M$ and $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$. Notice that all variables are values, and hence they can only represent computations if said computation is "thunked" (i.e. of the form $\{M\}$).

See Section 4.1 for an introduction into reduction frames.

**Definition 7.2** (CBPV reduction rules)**.** The beta reduction rules are defined as follows:

$$\begin{aligned} (\lambda x.M)V &\rightsquigarrow_{(\to)} M\,[V/x] \\ \{M\}! &\rightsquigarrow_{(U)} M \\ \texttt{let}\ x \leftarrow (\texttt{return}\ V)\ \texttt{in}\ N &\rightsquigarrow_{(F)} N\,[V/x] \end{aligned}$$

[1] This paper ignores the values $()$, $(V_1, V_2)$ and $\texttt{inj}_i V$, and the computations $\texttt{split}(V, x_1.x_2.M)$, $\texttt{case}_0(V)$, $\langle\rangle$, $\texttt{case}(V, x_1.M_1, x_2.M_2)$, $\langle M_1, M_2\rangle$ and $\texttt{prj}_i M$.

The reduction frames are defined as follows:

$$E ::= [\,] \mid EV \mid \texttt{let}\ E \leftarrow x\ \texttt{in}\ M$$

Along with the additional reduction rule:

$$\frac{M \rightsquigarrow M'}{E[M] \rightsquigarrow E[M']}$$

Notice that the only computations that have the potential to reduce are $MV$, $V!$ and $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$; $\lambda x.M$ and $\texttt{return}\ V$ cannot reduce by themselves.

## 7.2 Simple type system

There are two sorts of types for CBPV terms - values types and computation types.

$$\textit{(value types)}\ A, B ::= \phi \mid U\underline{\mathbf{A}}$$
$$\textit{(computation types)}\ \underline{\mathbf{A}}, \underline{\mathbf{B}} ::= A \to \underline{\mathbf{B}} \mid FA$$

Values are types with value types, computations with computation types.

**Definition 7.3** (Simple CBPV type assignments)**.** We take $\Gamma$ to be a map of variables to value types.

A typing judgement for a value has the form $\Gamma \vdash V : A$. Type assignments for values are:

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash M : \underline{\mathbf{A}}}{\Gamma \vdash \{M\} : U\underline{\mathbf{A}}}$$

A typing judgement for a computation has the form $\Gamma \vdash \underline{\mathbf{A}}$. Type assignments for computations are:

$$\frac{\Gamma, x : A \vdash M : \underline{\mathbf{B}}}{\Gamma \vdash \lambda x.M : A \to \underline{\mathbf{B}}}$$

$$\frac{\Gamma \vdash M : A \to \underline{\mathbf{B}} \qquad \Gamma \vdash V : A}{\Gamma \vdash MV : \underline{\mathbf{B}}}$$

$$\frac{\Gamma \vdash V : U\underline{\mathbf{A}}}{\Gamma \vdash V! : \underline{\mathbf{A}}}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{return}\ V : FA}$$

$$\frac{\Gamma \vdash M : FA \qquad \Gamma, x : A \vdash N : \underline{\mathbf{B}}}{\Gamma \vdash \texttt{let}\ x \leftarrow M\ \texttt{in}\ N : \underline{\mathbf{B}}}$$

## 7.3 Interpreting the $\lambda$ calculus

There are two different ways to interpret the lambda calculus - by modelling call-by-name reduction or modelling call-by-value reduction. The syntax of CBPV allows us to explicitly encode the execution order, so we will encode our chosen reduction strategy.

The original interpretations and proofs of Lemmas can be found in [13].

### 7.3.1 Call-by-name

#### Reduction simulation

See Section 4.2 for syntax, reduction and type system of the call-by-name subset of the $\lambda$-calculus.

In the call-by-name subset of the $\lambda$-calculus, the right-hand-side of an application is never evaluated. It is only evaluated when it is needed for the left-hand-side of an application. Hence, variables are always bound to unevaluated terms. If we think of call-by-name terms as CBPV computations, then terms are "thunked" when applied (or halted), and only resumed when a variable is evaluated (so a variable should be "forced").

Using this intuition, we can define our interpretation of call-by-name terms into CBPV.

**Definition 7.4** (Direct interpretation)**.** The interpretation operator

$$⟦\cdot⟧_n : \text{call-by-name terms} \to \text{CBPV computations}$$

is defined as:

$$\begin{aligned} ⟦x⟧_n &= x! \\ ⟦\lambda x.s⟧_n &= \lambda x.⟦s⟧_n \\ ⟦st⟧_n &= ⟦s⟧_n \, \{⟦t⟧_n\} \end{aligned}$$

**Example.**

$$\begin{aligned} ⟦(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))⟧_n &= (\lambda xy.y!) \, \{(\lambda x.x! \, \{x!\}) \, \{\lambda x.x! \, \{x!\}\}\} \\ &\rightsquigarrow \lambda y.y! \\ &= ⟦\lambda y.y⟧_n \end{aligned}$$

We would like that if $s \rightsquigarrow_n t$, then $⟦s⟧_n \rightsquigarrow ⟦t⟧_n$ (forwards simulation), and if $⟦s⟧_n \rightsquigarrow N$, then there exists a $t$ such that $⟦t⟧_n = N$ and $s \rightsquigarrow_n t$ (backwards simulation). However, there are edge cases that the "thunks" and "forces" cause.

**Example.** Example taken from [13, p. 6].

Consider the call-by-name term $(\lambda xy.x)z$.

$$(\lambda xy.x)z \rightsquigarrow_n \lambda y.z$$

$$⟦(\lambda xy.x)z⟧_n = (\lambda xy.x!) \, \{z!\}$$

$$⟦\lambda y.z⟧_n = \lambda y.z!$$

But

$$(\lambda xy.x!) \, \{z!\} \rightsquigarrow \lambda y. \, \{z!\}! \neq \lambda y.z!$$

This can be solved by allowing interpreted call-by-name terms an arbitrary amount of "force"-"thunk" pairs to surround them.

**Definition 7.5** (Simulation relation)**.** $\mapsto_n$ is a relation between call-by-name terms of the $\lambda$-calculus and CBPV computations. It is defined as:

$$s \mapsto_n \llbracket s \rrbracket_n$$

$$\frac{s \mapsto_n M}{s \mapsto_n \{M\}!}$$

**Lemma 7.6** (Forwards simulation)**.** *Let $s, t$ be call-by-name $\lambda$-terms. If $s \mapsto_n M$, and $s \rightsquigarrow_n^* t$, then there exists a CBPV term $N$ such that $t \mapsto_n N$ and $M \rightsquigarrow^* N$.*

**Lemma 7.7** (Backwards simulation)**.** *Let $s$ be a call-by-name $\lambda$-term. If $s \mapsto_n M$, and $M \rightsquigarrow^* N$, then there exists a call-by-name $\lambda$-term $t$ such that $t \mapsto_n N$ and $s \rightsquigarrow_n^* t$.*

### Type preservation

Since the type syntax is different, we need an interpretation of call-by-name $\lambda$-calculus types into CBPV types. All terms are interpreted as CBPV computations, so types should be interpreted as CBPV computation types.

We overload the same operator for type interpretation and environment interpretation.

**Definition 7.8** (Simple type interpretation)**.** The type interpretation operator

$$\cdot^n : \text{call-by-name types} \to \text{CBPV computation types}$$

is defined as:

$$\phi^n = F\phi$$
$$(A \to B)^n = UA^n \to B^n$$

The environment interpretation operator

$$\cdot^n : \text{call-by-name environments} \to \text{CBPV environments}$$

is defined as:

$$(x_1 : A_1, ..., x_n : A_n)^n = x_1 : UA_1^n, ..., x_n : UA_n^n$$

**Example.** We can type $\emptyset \vdash (\lambda xy.xy) : (A \to B) \to A \to B$.

$$\llbracket \lambda xy.xy \rrbracket_n = \lambda xy.x!\,\{y!\}$$

$$\dfrac{\dfrac{\dfrac{}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}})}}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash x! : U\underline{\mathbf{A}} \to \underline{\mathbf{B}}} \quad \dfrac{\dfrac{\dfrac{}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash y : U\underline{\mathbf{A}}}}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash y! : \underline{\mathbf{A}}}}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash \{y!\} : U\underline{\mathbf{A}}}}{\dfrac{\dfrac{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}), y : U\underline{\mathbf{A}} \vdash x!\,\{y!\} : \underline{\mathbf{B}}}{x : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}) \vdash \lambda y.x!\,\{y!\} : U\underline{\mathbf{A}} \to \underline{\mathbf{B}}}}{\emptyset \vdash \lambda xy.x!\,\{y!\} : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}) \to U\underline{\mathbf{A}} \to \underline{\mathbf{B}}}}$$

So we can type $\llbracket \lambda xy.xy \rrbracket_n : U(U\underline{\mathbf{A}} \to \underline{\mathbf{B}}) \to U\underline{\mathbf{A}} \to \underline{\mathbf{B}}$.

**Lemma 7.9** (Simple type preservation)**.** *Let $s$ be a call-by-name $\lambda$-term, and $M$ a CBPV computation. If $\Gamma \vdash s : A$, and $s \mapsto_n M$, then $\Gamma^n \vdash M : A^n$.*

### 7.3.2 Call by value

#### Reduction simulation

See Section 4.3 for syntax, reduction and type system of the call-by-value subset of the $\lambda$-calculus.

In call-by-value, the left-hand-side is first reduced to a value, then the right-hand side is reduced to a value, and only then application can occur. The $\texttt{let}\ \cdot \leftarrow \cdot\ \texttt{in}\ \cdot$ construct can be used to ensure that a particular subterm is reduced before anything else. So we can wrap call-by-value values in a $\texttt{return}$ to move forward in a reduction step.

Using this inutition, we can define our interpretation of call-by-name terms into CBPV.

**Definition 7.10** (Direct interpretation)**.** The interpretation operator

$$[\![\cdot]\!]_v : \text{call-by-value terms} \to \text{CBPV computations}$$

is defined as

$$\begin{aligned} [\![x]\!]_v &= \texttt{return}\ x \\ [\![\lambda x.s]\!]_v &= \texttt{return}\ \{\lambda x.[\![s]\!]_v\} \\ [\![st]\!]_v &= \texttt{let}\ f \leftarrow s\ \texttt{in let}\ v \leftarrow t\ \texttt{in}\ f!v \end{aligned}$$

**Example.**

$$\begin{aligned} [\![\lambda x.xx]\!]_v &= \texttt{return}\ \{\lambda x.\texttt{let}\ h \leftarrow \texttt{return}\ x\ \texttt{in let}\ z \leftarrow \texttt{return}\ x\ \texttt{in}\ h!z\} \\ &= \texttt{return}\ M \end{aligned}$$

$$\begin{aligned} [\![(\lambda x.xx)(\lambda x.xx)]\!]_v &= \texttt{let}\ g \leftarrow \texttt{return}\ M\ \texttt{in let}\ y \leftarrow \texttt{return}\ M\ \texttt{in}\ g!y \\ &= N \end{aligned}$$

$$\begin{aligned} [\![(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))]\!]_v &= \texttt{let}\ f \leftarrow (\texttt{return}\ \{\lambda x.\texttt{return}\ \{\lambda y.\texttt{return}\ y\}\})\ \texttt{in let}\ x \leftarrow N\ \texttt{in}\ f!x \\ &= O \end{aligned}$$

$$\begin{aligned} N &\rightsquigarrow \texttt{let}\ y \leftarrow \texttt{return}\ M\ \texttt{in}\ M!y \\ &\rightsquigarrow M!M \\ &\rightsquigarrow (\lambda x.\texttt{let}\ h \leftarrow \texttt{return}\ x\ \texttt{in let}\ z \leftarrow \texttt{return}\ x\ \texttt{in}\ h!z)M \\ &\rightsquigarrow \texttt{let}\ h \leftarrow \texttt{return}\ M\ \texttt{in let}\ z \leftarrow \texttt{return}\ M\ \texttt{in}\ h!z \\ &= N \end{aligned}$$

$$\begin{aligned} O &\rightsquigarrow \texttt{let}\ x \leftarrow N\ \texttt{in}\ \{\texttt{return}\ \lambda x.\texttt{return}\ \lambda y.\texttt{return}\ y\}!x \\ &\rightsquigarrow^* \texttt{let}\ x \leftarrow N\ \texttt{in}\ \{\texttt{return}\ \lambda x.\texttt{return}\ \lambda y.\texttt{return}\ y\}!x \\ &\rightsquigarrow^* .. \end{aligned}$$

**Lemma 7.11** (Forwards simulation)**.** *Let $s, t$ be call-by-value $\lambda$-terms. If $s \rightsquigarrow_v^* t$, then $[\![s]\!]_v \rightsquigarrow^* [\![t]\!]_v$.*

**Lemma 7.12** (Backwards simulation)**.** *Let $s$ be a call-by-value $\lambda$-term. If $[\![s]\!]_v \rightsquigarrow^* N$, then there exists a call-by-value $\lambda$-term $t$ such that $[\![t]\!]_v = N$ and $s \rightsquigarrow_v^* t$.*

#### Type preservation

Similar to call-by-name, we need an interpretation of call-by-value $\lambda$-calculus types into CBPV types. Since call-by-value functions are interpreted as CBPV values, the $\lambda$-calculus arrow type must be

interpreted into a CBPV value type.

We overload the same operator for type interpretation and environment interpretation.

**Definition 7.13** (Simple type interpretation)**.** The type interpretation operator

$$\cdot^v : \text{call-by-value types} \to \text{CBPV value types}$$

is defined as:

$$\phi^v = \phi$$
$$(A \to B)^v = U(A^v \to FB^v)$$

The environment interpretation operator

$$\cdot^v : \text{call-by-value environments} \to \text{CBPV environments}$$

is defined as:

$$(x_1 : A_1, ..., x_n : A_n)^v = x_1 : A_1^v, ..., x_n : A_n^v$$

**Lemma 7.14** (Simple type preservation)**.** *Let $v$ be a call-by-value value. If $\Gamma \vdash_v v : A$ then $\Gamma^v \vdash [\![v]\!]_v : A^v$.*

*Let $s$ be a call-by-value term. If $\Gamma \vdash_s s : A$ then $\Gamma^v \vdash [\![s]\!]_v : FA^v$.*

# Chapter 8

# Abstract Machines

Abstract (stack) machines are a useful way to represent operational semantics of a calculus. They consist of a term and a context (see Section 4.1) in which the term is run:

$$\langle s \mid E \rangle$$

(we use the same syntax as $\bar{\lambda}\mu\tilde{\mu}$ commands for machines). This machine can be viewed as a deconstruction of the term $E[s]$.

Machines can run via rules that are specified, usually to reduce the term on the left-hand-side.

## 8.1 $\lambda$-calculus machine rules

The $\lambda$-calculus can be run with two different strategies: call-by-name and call-by-value (see Chapter 4). Since the reductions are different, their operational semantics and hence stack rules are different.

For simplicity, we use $s \cdot E$ to mean the context $E[[\,]s]$.

### 8.1.1 Call-by-name

**Definition 8.1** (Call-by-name machine rules)**.** In call-by-name $\lambda$-calculus, the machine rules are:

$$\langle st \mid E \rangle \rightsquigarrow \langle s \mid t \cdot E \rangle$$
$$\langle \lambda x.s \mid t \cdot E \rangle \rightsquigarrow \langle s\,[t/x] \mid E \rangle$$

This is very easily justifiable - in a call-by-name application $st$, you reduce $s$ until it becomes a $\lambda$ expression, and then apply $t$.

### 8.1.2 Call-by-value

In call-by-value $\lambda$-calculus, the machine rules are trickier. In an application $st$, once $s$ has finished reducing to a value, you then must reduce $t$ to a value before applying. A first thought might be to define the rules as so:

$$\begin{aligned}\langle st \mid E\rangle &\rightsquigarrow \langle s \mid t \cdot E\rangle\\ \langle v \mid t \cdot E\rangle &\rightsquigarrow \langle t \mid v \cdot E\rangle\\ \langle \lambda x.s \mid v \cdot E\rangle &\rightsquigarrow \langle s\,[^v/_x] \mid E\rangle\end{aligned}$$

But this is incorrect: $y(\lambda x.x)$ will be run as so:

$$\begin{aligned}&\langle y(\lambda x.x) \mid E\rangle\\ \rightsquigarrow &\langle y \mid (\lambda x.x) \cdot E\rangle\\ \rightsquigarrow &\langle \lambda x.x \mid y \cdot E\rangle\\ \rightsquigarrow &\langle y \mid E\rangle\end{aligned}$$

$z(\lambda x.x)$ is in normal form and so shouldn't reduce, especially not to $y$. This error is because in the second rule, the context that $t$ is running in shouldn't be $v \cdot E = E[[\ ]v]$, it should be $E[v[\ ]]$. For simplicity, we will use $v \circ E$ to mean the context $E[v[\ ]]$.

**Definition 8.2** (Call-by-value machine rules)**.** In call-by-value $\lambda$-caluculus, the machine rules are:

$$\begin{aligned}\langle st \mid E\rangle &\rightsquigarrow \langle s \mid t \cdot E\rangle\\ \langle v \mid t \cdot E\rangle &\rightsquigarrow \langle t \mid v \circ E\rangle\\ \langle v \mid (\lambda x.s) \circ E\rangle &\rightsquigarrow \langle s\,[^v/_x] \mid E\rangle\end{aligned}$$

Notice that the rule for running an application in any context is the same for both call-by-name and call-by-value.

## 8.2 $\bar{\lambda}\mu\tilde{\mu}$ as an abtract machine

The syntax from the previous section is very similar to that of $\bar{\lambda}\mu\tilde{\mu}$'s syntax (see Section 6.2). We can in fact build the remaining structures of $\bar{\lambda}\mu\tilde{\mu}$ from these rules, and define the reduction rules of $\bar{\lambda}\mu\tilde{\mu}$ using the above stack rules. This section takes inspiration from [5].

The term $st$ and context $v \circ E$ do not appear in $\bar{\lambda}\mu\tilde{\mu}$ syntax, so we need a way to represent them. We also have a clashing machine reduction rule for $\langle \lambda x.s \mid v \cdot E\rangle$.

Consider the call-by-name/call-by-value rule for running an application $st$:

$$\langle st \mid E\rangle \rightsquigarrow \langle s \mid t \cdot E\rangle$$

We can view the term $st$ as a function which takes a context $E$ and returns the machine $\langle s \mid t \cdot E\rangle$, where the function itself is a term. We do not have a symbol that represents this kind of function signature: let's use $\mu$. Since we are taking contexts, it would be wise to use a different set of variables to latin characters, which typically represent terms. Let's use greek variables $\alpha, \beta, ....$

So $st$ can be represented as the term $\mu\alpha.\langle s \mid t \cdot \alpha\rangle$. Notice that this is exactly how the term $st$ is translated into $\bar{\lambda}\mu\tilde{\mu}$ from Section 6.4.

Consider the following call-by-value rule for running a value in the context $v \circ E$:

$$\langle v \mid (\lambda x.s) \circ E\rangle \rightsquigarrow \langle s\,[^v/_x] \mid E\rangle$$

We can view the context $(\lambda x.s) \circ E$ as a function which takes a term $v$ and returns the machine $\langle s\,[^v/_x] \mid E\rangle$, where the function itself is a context. We do not have a symbol that represents this kind of function signature: let's use $\tilde{\mu}$. Since we are taking terms, we can use latin variables $x, y, ....$

So $(\lambda x.s) \circ E$ can be represented as the context $\tilde{\mu}y.\langle s\,[^y/_x] \mid E\rangle$. Since $s\,[^y/_x]$ is just a re-naming of variables, this is the same context as $\tilde{\mu}x.\langle s \mid E\rangle$.

Between call-by-name and call-by-value, the rule for beta reduction $\langle \lambda x.s \mid t \cdot E\rangle$ differs.

In call-by-name, the rules give the reduction:

$$\langle \lambda x.s \mid t \cdot E\rangle \rightsquigarrow \langle s\,[^t/_x] \mid E\rangle$$

However, in call-by-value the rules give the reduction:

$$\begin{aligned}&\langle \lambda x.s \mid t \cdot E\rangle\\ \rightsquigarrow &\langle t \mid (\lambda x.s) \circ E\rangle = \langle t \mid \tilde{\mu}x.\langle s \mid E\rangle\rangle\end{aligned}$$

We cannot represent call-by-value reduction via the first call-by-name-style rule, as it would completely remove the chance to ever reduce $t$ before it is applied. But call-by-name reduction can be modelled by the second call-by-value-style rule by applying the function $\tilde{\mu}x$ first:

$$\begin{aligned}&\langle \lambda x.s \mid t \cdot E\rangle\\ \rightsquigarrow &\langle t \mid \tilde{\mu}x.\langle s \mid E\rangle\rangle\\ \rightsquigarrow &\langle s\,[^t/_x] \mid E\rangle\end{aligned}$$

So we can reach the final rules for $\bar{\lambda}\mu\tilde{\mu}$-reduction:

$$\begin{aligned}\langle \lambda x.s \mid t \cdot E\rangle &\rightsquigarrow_{\beta} \langle t \mid \tilde{\mu}x.\langle s \mid E\rangle\rangle\\ \langle \mu\alpha.c \mid E\rangle &\rightsquigarrow_{(\mu)} c\,[^E/_\alpha]\\ \langle s \mid \tilde{\mu}x.c\rangle &\rightsquigarrow_{(\tilde{\mu})} c\,[^s/_x]\end{aligned}$$

## 8.3 Call By Push Value machine rules

In the same vein as the $\lambda$-calculus, we can represent the operational semantics of CBPV in a machine. Levy included the operational semantics of CBPV in a stack machine in [7, p. 47]. Here we translate these into the $\bar{\lambda}\mu\tilde{\mu}$-style machines.

As with the $\lambda$-calculus, we use $V \cdot E$ to represent the context $E[[\,]V]$ (or in Levy's syntax, $V :: E$).

**Definition 8.3** (CBPV machine rules)**.** In CBPV, the machine rules are:

$$\begin{aligned}\langle MV \mid E\rangle &\rightsquigarrow \langle M \mid V \cdot E\rangle\\ \langle \lambda x.M \mid V \cdot E\rangle &\rightsquigarrow \langle M\,[^V/_x] \mid E\rangle\\ \langle \texttt{let } x \leftarrow M \texttt{ in } N \mid E\rangle &\rightsquigarrow \langle M \mid E[\texttt{let } x \leftarrow [\,] \texttt{ in } N]\rangle\\ \langle \texttt{return } V \mid E[\texttt{let } x \leftarrow [\,] \texttt{ in } N]\rangle &\rightsquigarrow \langle N\,[^V/_x] \mid E\rangle\\ \langle \{M\}! \mid E\rangle &\rightsquigarrow \langle M \mid E\rangle\end{aligned}$$

# Chapter 9

# Call by Push Value in $\bar{\lambda}\mu\tilde{\mu}$

We can interpret Call by Push Value into the $\bar{\lambda}\mu\tilde{\mu}$-calculus and simulate it via the $\bar{\lambda}\mu\tilde{\mu}$ reduction rules.

## 9.1 $\bar{\lambda}\mu\tilde{\mu}$ as a CBPV stack

In Section 8.2, we translated the $\lambda$-calculus terms and contexts into $\bar{\lambda}\mu\tilde{\mu}$'s syntax and derived the reduction rules from the call-by-name and call-by-value stack rules. We can do the same for the CBPV stack rules.

### 9.1.1 Application

Consider the stack rule for $\langle MV \mid E\rangle$ from Definition 8.3:

$$\begin{aligned}&\langle MV \mid E\rangle\\ \rightsquigarrow\ &\langle M \mid V \cdot E\rangle\end{aligned}$$

This mimics how call-by-name and call-by-value stack rules deal with application (Definitions 8.1 and 8.2), so we can interpret it similarly: $MV$ is a function that takes a context $E$ and returns the stack $\langle M \mid V \cdot E\rangle$, so it can be represented by the term $\mu\alpha.\langle M \mid V \cdot \alpha\rangle$.

### 9.1.2 Lambda expressions

Consider the stack rule for $\langle \lambda x.M \mid V \cdot E\rangle$ from Definition 8.3:

$$\begin{aligned}&\langle \lambda x.M \mid V \cdot E\rangle\\ \rightsquigarrow\ &\langle M\,[V/x] \mid E\rangle\end{aligned}$$

This can be considered as a restriction of $\bar{\lambda}\mu\tilde{\mu}$'s reduction rules: $\langle \bar{\lambda}x.v_1 \mid v_2 \cdot e\rangle \rightsquigarrow \langle v_1\,[v_2/x] \mid e\rangle$. Hence $\lambda x.M$ can be interpreted as $\bar{\lambda}x.M$.

### 9.1.3 Let-in expressions and returns

Consider the stack rules for $\langle \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \mid E\rangle$ from Definition 8.3:

$$\begin{aligned} &\langle \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \mid E\rangle \\ \rightsquigarrow &\langle M \mid E[\texttt{let}\ x \leftarrow [\,]\ \texttt{in}\ N]\rangle \end{aligned}$$

and

$$\begin{aligned} &\langle \texttt{return}\ V \mid E[\texttt{let}\ x \leftarrow [\,]\ \texttt{in}\ N]\rangle \\ \rightsquigarrow &\langle N\,[^V/_x] \mid E\rangle \end{aligned}$$

From the second rule, we can view $E[\texttt{let}\ x \leftarrow [\,]\ \texttt{in}\ N]$ as a function from terms to stacks, that is itself a context. From Section 8.2, this corresponds to the signature of $\tilde{\mu}$. It takes a term $\texttt{return}\ V$ and returns the stack $\langle N\,[^V/_x] \mid E\rangle$.

The $\texttt{return}$ isn't doing anything interesting in this scenario - it only halts further computation of its contents and progresses the reduction of the $\texttt{let}\ \cdot \leftarrow \cdot\ \texttt{in}\ \cdot$ expression. Since in CBPV values and computations are syntactically distinct, we can pattern match on the $V$ of the left-hand-side of the stack instead of the $\texttt{return}$ .

So $\texttt{return}\ V$ can be interpreted as solely $V$, and hence $\texttt{let}\ x \leftarrow [\,]\ \texttt{in}\ N$ can be interpreted as $\tilde{\mu}x.\langle N \mid E\rangle$.

Now $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$ can be considered as a function from contexts to stacks, that is itself a term. From Section 8.2, this corresponds to the signature of $\mu$. It maps the context $E$ to the stack $\langle M \mid E[\texttt{let}\ x \leftarrow [\,]\ \texttt{in}\ N]\rangle = \langle M \mid \tilde{\mu}x.\langle N \mid E\rangle\rangle$, so can be interpreted as the term $\mu\alpha.\langle M \mid \tilde{\mu}x.\langle N \mid \alpha\rangle\rangle$.

### 9.1.4 Forces and thunks

Consider the stack rules for force $(\cdot!)$ and thunk $(\{\cdot\})$:

$$\begin{aligned} &\langle \{M\}! \mid E\rangle \\ \rightsquigarrow &\langle M \mid E\rangle \end{aligned}$$

So the term $\{M\}!$ is a function that takes a context $E$ and returns the stack $\langle M \mid E\rangle$. So it can be represented as the function $\mu\alpha.\langle M \mid \alpha\rangle \rightsquigarrow_{(\eta\mu)} M$. By itself, it appears that $\cdot!$ and $\{\cdot\}$ do nothing: they don't alter the stack in any way, so they can be gotten rid of it seems.

But consider the CBPV computation $\texttt{let}\ x \leftarrow y!\ \texttt{in}\ N$. This is in normal form in CBPV, and so has no reductions. But, translated into our $\bar{\lambda}\mu\tilde{\mu}$-style stack this would give:

$$\begin{aligned} &\langle \mu\alpha.\langle y \mid \tilde{\mu}x.\langle N \mid \alpha\rangle\rangle \mid [\,]\rangle \\ \rightsquigarrow &\langle y \mid \tilde{\mu}x.\langle N \mid [\,]\rangle\rangle \\ \rightsquigarrow &\langle N \mid [\,]\rangle \end{aligned}$$

So our translation tells us that in fact $\texttt{let}\ x \leftarrow y!\ \texttt{in}\ N \rightsquigarrow N$, which is incorrect.

The problem lies in the activation of the $\tilde{\mu}x$ - in the previous subsection, we restricted $\tilde{\mu}$ to run if and only if the left-hand-side was a value. Since we removed the ! label, it became the value $y$, and so the $\tilde{\mu}$ was activated.

So we cannot have $\cdot!$ and $\{\cdot\}$ absent from our translation.

We propose another way of viewing $\cdot!$ and $\{\cdot\}$ in a stack:

$$\begin{aligned} &\langle V! \mid E\rangle \\ \rightsquigarrow &\langle V \mid E[[\,]!]\rangle \end{aligned}$$

$$\begin{aligned} &\langle \{M\} \mid E[[\,]!]\rangle \\ \rightsquigarrow &\langle M \mid E\rangle \end{aligned}$$

We essentially "push" $[\,]!$ onto the stack, and use $\{\cdot\}$ to remove it and proceed. Since these constructs don't interact with any other construct in CBPV, we propose to carry forward the notation from CBPV into $\bar{\lambda}\mu\tilde{\mu}$. We propose the syntax $! \circ E$ for the context $E[[\,]!]$, so $V!$ would be translated as $\mu\alpha.\langle C|! \circ \alpha\rangle$. $\{M\}$ remains the same syntax.

## 9.2 Interpretation

Based on the previous section, we formalise our interpretation.

First, we augment the syntax of $\bar{\lambda}\mu\tilde{\mu}$ with $\{\cdot\}$ and $!$.

**Definition 9.1** ($\bar{\lambda}\mu\tilde{\mu}$ augmented syntax)**.** The augmented $\bar{\lambda}\mu\tilde{\mu}$ syntax is defined as so:

$$\begin{aligned} \mathbf{v} &::= x \mid \{v\} \\ \text{(values)}\quad v &::= \mathbf{v} \mid \mu\alpha.c \mid \bar{\lambda}x.v \\ \text{(commands)}\quad c &::= \langle v \mid e\rangle \\ \text{(contexts)}\quad e &::= \alpha \mid \tilde{\mu}x.c \mid v \cdot e|! \circ e \end{aligned}$$

We add a restriction of commands:

$$\begin{aligned} \mathbf{c} ::= &\langle x|! \circ e\rangle \mid \langle x \mid v \cdot e\rangle \\ &\mid \langle \{v\} \mid v \cdot e\rangle \\ &\mid \langle \bar{\lambda}x.v|! \circ e\rangle \mid \langle \bar{\lambda}x.v \mid \tilde{\mu}x.c\rangle \end{aligned}$$

$\mathbf{v}$ is used to represent the values that CBPV values will be translated into, in order to restrict reduction accordingly (such as only activating $\rightsquigarrow_{(\tilde{\mu})}$ when the left-hand-side of the command is a CBPV value). All other values are potentially what CBPV copmutations can be translated into. Any computation that (could potentially be) redex will be translated into the form $\mu\alpha.c$, as to reduce it must involve some manipulation of the corresponding context.

$\mathbf{c}$ represents all commands that are in normal form that do not have the context $\alpha$, given the reduction rules below. This is used to restrict $\rightsquigarrow_{(\mu)}$ to $\langle \mu\alpha.\mathbf{c} \mid e\rangle \rightsquigarrow_{(\mu)} \mathbf{c}\,[^e/_\alpha]$. Say we included $\alpha$ in the potential contexts of $\mathbf{c}$ - then we would have a choice of reducing $\rightsquigarrow_{(\mu)}$ or $\rightsquigarrow_{(\eta\mu)}$:

$$\begin{aligned} \langle \mu\alpha.\langle v \mid \alpha\rangle \mid e\rangle &\rightsquigarrow_{(\mu)} \langle v\,[^e/_\alpha] \mid e\rangle \\ &= \langle v \mid e\rangle \qquad\qquad \alpha \notin v \end{aligned}$$

$$\langle \mu\alpha.\langle v \mid \alpha\rangle \mid e\rangle \rightsquigarrow_{(\eta\mu)} \langle v \mid \alpha\rangle \qquad \alpha \notin v$$

For our purposes of translating CBPV, the variable $\alpha$ appears only once in any term, so these reductions are the same and $\rightsquigarrow_{(\eta\mu)}$ removes the need for $\alpha$ in the context.

**Definition 9.2** (Restricted $\bar{\lambda}\mu\tilde{\mu}$ reduction)**.** The restricted reduction frames are defined as follows:

$$E = [\,] \mid \langle E \mid e \rangle \mid \mu\alpha.E$$

The restricted reduction rules are defined as follows:

$$\begin{aligned}
\langle \bar{\lambda}x.v_1 \mid v_2 \cdot e \rangle &\rightsquigarrow_{\beta} \langle v_1 \, [^{v_2}/_x] \mid e \rangle \\
\langle \mu\alpha.\mathbf{c} \mid e \rangle &\rightsquigarrow_{(\mu)} \mathbf{c} \, [^{e}/_{\alpha}] \\
\langle \mathbf{v} \mid \tilde{\mu}x.c \rangle &\rightsquigarrow_{(\tilde{\mu})} c \, [^{V}/_{x}] \\
\langle \{v\} \mid ! \circ e \rangle &\rightsquigarrow_{(U)} \langle v \mid e \rangle \\
\mu\alpha. \langle v \mid \alpha \rangle &\rightsquigarrow_{(\eta\mu)} v
\end{aligned}$$

$$\frac{c \rightsquigarrow c'}{E[c] \rightsquigarrow E[c']}$$

And now our interpretation.

**Definition 9.3** (Direct interpretation)**.** The interpretation operator

$$[\![\cdot]\!] : CBPV \text{ terms} \to \bar{\lambda}\mu\tilde{\mu} \text{ values}$$

is defined as:

$$\begin{aligned}
[\![x]\!] &= x \\
[\![\{M\}]\!] &= \{[\![M]\!]\} \\
[\![V!]\!] &= \mu\alpha. \langle [\![V]\!] \mid ! \circ \alpha \rangle \\
[\![\lambda x.M]\!] &= \bar{\lambda}x.[\![M]\!] \\
[\![MV]\!] &= \mu\alpha. \langle [\![M]\!] \mid [\![V]\!] \cdot \alpha \rangle \\
[\![\texttt{let } x \leftarrow M \texttt{ in } N]\!] &= \mu\alpha. \langle [\![M]\!] \mid \tilde{\mu}x. \langle [\![N]\!] \mid \alpha \rangle \rangle \\
[\![\texttt{return } V]\!] &= [\![V]\!]
\end{aligned}$$

We need to be able to reduce under the first $\mu$ abstraction, since CBPV computations that could potentially reduce (such as $MV$) are translated to a $\mu$ abstraction. So the reduction frame $\mu\alpha.[\,]$ is necessary in order to prove simulation. Notice that the context $\mu\alpha. \langle \mu\beta.[\,] \mid e \rangle$ is not included in our specified reduction frames - you can only reduce underneath a single, outside $\mu$ abtraction.

$\rightsquigarrow_{(\eta\mu)}$ is also needed in order to prove simualation, similar to how it is used to prove preservation of $\beta$-reduction for the $\lambda$-calculus (see Lemma 6.10).

Below we outline some Lemmas that are useful in understanding the interpretation.

**Lemma 9.4** (CBPV values and computations are distinct)**.** *Let $V$ be a CBPV value. Then $[\![V]\!] \in \mathbf{v}$.*

*Let $M$ be a CBPV computation, $M \neq$ `return` $V$. Then $[\![M]\!] \notin \mathbf{v}$.*

*Proof.* Let $V$ be a CBPV value. Either $V = x$ or $V = \{M\}$ for some computation $M$. So $[\![V]\!] = x \in \mathbf{v}$ or $[\![V]\!] = \{[\![M]\!]\}$. By definition, $[\![M]\!] \in v$, so $\{[\![M]\!]\} = [\![V]\!] \in \mathbf{v}$.

Let $M$ be a CBPV computation. Either $M = V!$, $M = \lambda x.M'$, $M = M'V$, or $M = \texttt{let } x \leftarrow M' \texttt{ in } N$.

Suppose $M = V!$. $[\![M]\!] = \mu\alpha. \langle [\![V]\!] \mid ! \circ \alpha \rangle \notin \mathbf{v}$.

Suppose $M = \lambda x.M'$. $[\![M]\!] = \bar{\lambda}x.[\![M']\!] \notin \mathbf{v}$.

Suppose $M = M'V$. $[\![M]\!] = \mu\alpha.\langle [\![M']\!] \mid V \cdot \alpha\rangle \notin \mathbf{v}$.

Suppose $M = \texttt{let } x \leftarrow M' \texttt{ in } N$. $[\![M]\!] = \mu\alpha.\langle [\![M']\!] \mid \tilde{\mu}x.\langle [\![N]\!] \mid \alpha\rangle\rangle \notin \mathbf{v}$. □

**Lemma 9.5** (Only application, force and let-in reduce)**.** *Let $M$ be a CBPV computation. If there exists a CBPV computation $M'$ such that $M \rightsquigarrow M'$, then $M$ is either of the form: $NV$, $V!$ or $\texttt{let } x \leftarrow N \texttt{ in } O$.*

*Proof.* See Definition 7.2 for the reduction rules of CBPV.

Suppose $\rightsquigarrow$ is a primitive reduction.

If $\rightsquigarrow = \rightsquigarrow_\beta$, then $M = (\lambda x.N')V$ for some $N', V$. Hence $M$ is of the form $NV$.

If $\rightsquigarrow = \rightsquigarrow_{(U)}$, then $M = \{N\}!$ for some $N$. Hence $M$ is of the form $V!$.

If $\rightsquigarrow = \rightsquigarrow_{(F)}$, then $M = \texttt{let } x \leftarrow \texttt{return } V \texttt{ in } O$ for some $V, O$. Hence $M$ is of the form $\texttt{let } x \leftarrow N \texttt{ in } O$.

Suppose $\rightsquigarrow$ is not a primitive reduction, so $M = E[N]$ for some reduction frame $E$ and some computation $N$.

Let $E = [\,]$. Then $M \rightsquigarrow M'$ is a primitive reduction and this holds trivially.

Let $E = E'V$. Then $M = E[N] = (E'[N])V$, so $M$ is of the form $NV$.

Let $E = \texttt{let } x \leftarrow E' \texttt{ in } O$. Then $M = E[N] = \texttt{let } x \leftarrow E'[N] \texttt{ in } O$, so $M$ is of the form $\texttt{let } x \leftarrow N \texttt{ in } O$. □

**Lemma 9.6** (CBPV redexes are $\mu$ abstractions)**.** *Let $M$ be a CBPV computation. If there exists a CBPV computation $M'$ such that $M \rightsquigarrow M'$, then $[\![M]\!] = \mu\alpha.c$ for some $\bar{\lambda}\mu\tilde{\mu}$ command $c$, and $\alpha$ appears exactly once in $c$.*

*Proof.* By Lemma 9.5, $M$ must be either $NV$, $V!$ or $\texttt{let } x \leftarrow N \texttt{ in } O$, as $M \rightsquigarrow M'$. From our interpretation we have

$$[\![NV]\!] = \mu\alpha.\langle [\![N]\!] \mid V \cdot \alpha\rangle$$
$$[\![V!]\!] = \mu\alpha.\langle [\![V]\!] \mid ! \circ \alpha\rangle$$
$$[\![\texttt{let } x \leftarrow N \texttt{ in } O]\!] = \mu\alpha.\langle [\![N]\!] \mid \tilde{\mu}x.\langle [\![O]\!] \mid \alpha\rangle\rangle$$

Hence $[\![M]\!] = \mu\alpha.c$ for some command $c$.

The translations of all CBPV terms are compostitional - they do not rely on any surrounding term. In particular, they all introduce their own variables, if any. So it is not possible to have any reference to $\alpha$ inside the translation of a subterm of a term.

All possible translations of a redex $M$ have only one reference to $\alpha$, as displayed above, hence $\alpha$ appears only once in $c$ if $[\![M]\!] = \mu\alpha.c$. □

Now our final results.

**Proposition 9.7** (Forward simulation of CBPV in $\bar{\lambda}\mu\tilde{\mu}$)**.** Let $M, M'$ be CBPV computations. If $M \rightsquigarrow M'$, then $[\![M]\!] \rightsquigarrow [\![M']\!]$.

*Proof.* We will first consider each primitive reduction relation of CBPV (see Definition 7.2 for the reduction relations of CBPV).

$$\begin{aligned}
\llbracket(\lambda x.M)V\rrbracket &= \mu\alpha.\,\langle\llbracket\lambda x.M\rrbracket \mid \llbracket V\rrbracket\cdot\alpha\rangle \\
&= \mu\alpha.\,\langle\bar{\lambda}x.\llbracket M\rrbracket \mid \llbracket V\rrbracket\cdot\alpha\rangle \\
&\rightsquigarrow_{\beta} \mu\alpha.\,\langle\llbracket M\rrbracket\,[\llbracket V\rrbracket/x] \mid \alpha\rangle \\
&\rightsquigarrow_{(\eta\mu)} \llbracket M\rrbracket\,[\llbracket V\rrbracket/x] \\
&= \llbracket M\,[V/x]\rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket\{M\}!\rrbracket &= \mu\alpha.\,\langle\llbracket\{M\}\rrbracket \mid !\circ\alpha\rangle \\
&= \mu\alpha.\,\langle\{\llbracket M\rrbracket\} \mid !\circ\alpha\rangle \\
&\rightsquigarrow_{(U)} \mu\alpha.\,\langle\llbracket M\rrbracket \mid \alpha\rangle \\
&\rightsquigarrow_{(\eta\mu)} \llbracket M\rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket\texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ M\rrbracket &= \mu\alpha.\,\langle\llbracket\texttt{return}\ V\rrbracket \mid \tilde{\mu}x.\,\langle\llbracket M\rrbracket \mid \alpha\rangle\rangle \\
&= \mu\alpha.\,\langle\llbracket V\rrbracket \mid \tilde{\mu}x.\,\langle\llbracket M\rrbracket \mid \alpha\rangle\rangle \\
&\rightsquigarrow_{(\tilde{\mu})} \mu\alpha.\,\langle\llbracket M\rrbracket\,[\llbracket V\rrbracket/x] \mid \alpha\rangle && \text{by Lemma 9.7} \\
&\rightsquigarrow_{(\eta\mu)} \llbracket M\rrbracket\,[\llbracket V\rrbracket/x] \\
&= \llbracket M\,[V/x]\rrbracket
\end{aligned}$$

We now consider the reduction rule:

$$\frac{M \rightsquigarrow M'}{E[M] \rightsquigarrow E[M']}$$

along with the reduction frames:

$$E = [\,] \mid EV \mid \texttt{let}\ x \leftarrow E\ \texttt{in}\ M$$

Assume $\llbracket M\rrbracket \rightsquigarrow \llbracket M'\rrbracket$. We will induct over the structure of the reduction frames $E$.

Let $E = [\,]$. Then $\llbracket M\rrbracket \rightsquigarrow \llbracket M'\rrbracket$ holds trivially.

Let $E = E'V$, for some reduction frame $E'$. Assume that $\llbracket E'[M]\rrbracket \rightsquigarrow \llbracket E'[M']\rrbracket$.

$$\begin{aligned}
\llbracket E[M]\rrbracket &= \llbracket E'[M]V\rrbracket \\
&= \mu\alpha.\,\langle\llbracket E'[M]\rrbracket \mid \llbracket V\rrbracket\cdot\alpha\rangle
\end{aligned}$$

We have the reduction frame $\mu\alpha.\langle [\,] \mid \llbracket V \rrbracket \cdot \alpha\rangle$ in our augmented $\bar{\lambda}\mu\tilde{\mu}$ definition. Unless $\llbracket E'[M] \rrbracket \rightsquigarrow \llbracket E'[M'] \rrbracket$ runs over $\bar{\lambda}x.v$, we are done. But $\bar{\lambda}x.v$ does not reduce - there are no primitive reductions starting from it and no reduction frames to allow reduction underneath the abstraction. Hence

$$\begin{aligned}\mu\alpha.\langle \llbracket E'[M] \rrbracket \mid \llbracket V \rrbracket \cdot \alpha\rangle &\rightsquigarrow \mu\alpha.\langle \llbracket E'[M'] \rrbracket \mid \llbracket V \rrbracket \cdot \alpha\rangle \\ &= \llbracket E'[M']V \rrbracket \\ &= \llbracket E[M'] \rrbracket\end{aligned}$$

Let $E = \texttt{let}\ \ x \leftarrow E'\ \ \texttt{in}\ \ N$, Assume that $\llbracket E'[M] \rrbracket \rightsquigarrow \llbracket E'[M'] \rrbracket$.

$$\begin{aligned}\llbracket E[M] \rrbracket &= \llbracket \texttt{let}\ \ x \leftarrow E'[M]\ \ \texttt{in}\ \ N \rrbracket \\ &= \mu\alpha.\langle \llbracket E'[M] \rrbracket \mid \tilde{\mu}x.\langle \llbracket N \rrbracket \mid \alpha\rangle\rangle\end{aligned}$$

Again, the reduction frame $\mu\alpha.\langle [\,] \mid \tilde{\mu}x.\langle \llbracket N \rrbracket \mid \alpha\rangle\rangle$ is in our augmented $\bar{\lambda}\mu\tilde{\mu}$ definition. The only way reduction could differ is if $\llbracket E'[M] \rrbracket \rightsquigarrow \llbracket E'[M'] \rrbracket$ runs over a value in $\mathbf{v}$. But as with the previous case, no $\mathbf{v}$ can reduce via any primitive reduction or reduction frame. So

$$\begin{aligned}\mu\alpha.\langle \llbracket E'[M] \rrbracket \mid \tilde{\mu}x.\langle \llbracket N \rrbracket \mid \alpha\rangle\rangle &\rightsquigarrow \mu\alpha.\langle \llbracket E'[M'] \rrbracket \mid \tilde{\mu}x.\langle \llbracket N \rrbracket \mid \alpha\rangle\rangle \\ &= \llbracket \texttt{let}\ \ x \leftarrow E'[M']\ \ \texttt{in}\ \ N \rrbracket \\ &= \llbracket E[M'] \rrbracket\end{aligned}$$

□

**Proposition 9.8** (Backwards simulation of CBPV)**.** Let $M, M'$ be CBPV values. If $\llbracket M \rrbracket \rightsquigarrow \llbracket M' \rrbracket$, then $M \rightsquigarrow M'$.

This paper gives the background for how this Proposition works in Section 9.1, but does not give the proof.

# Chapter 10

# Conclusion

We have given an interpretation of CBPV into an augmented $\bar{\lambda}\mu\tilde{\mu}$-calculus. The background of this paper gives the intuition behind $\bar{\lambda}\mu\tilde{\mu}$ as a machine to run both the call-by-name and call-by-value $\lambda$-calculus reduction strategies, which is then applied to the subsuming paradigm CBPV.

## 10.1 Drawbacks

We had to augment the $\bar{\lambda}\mu\tilde{\mu}$-calculus in order to simulate CBPV. Artificially "thunking" and "forcing" computation cannot be re-produced in $\bar{\lambda}\mu\tilde{\mu}$, since these constructs don't interact with any context they're running in. We could've used $\bar{\lambda}\mu\tilde{\mu}$'s $\tilde{\mu}$ symbol to represent "force", and restricted reduction of it to run only when the left-hand-side was the interpretation of $\{M\}$. This would mean that the $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$ and $\texttt{return}\ V$ constructs would need to be translated differently. But we thought that the translation of $\texttt{let}\ x \leftarrow M\ \texttt{in}\ N$ as $\mu\alpha.\langle M \mid \tilde{\mu}x.\langle N \mid \alpha\rangle\rangle$ was faithful to the method used to generate the $\bar{\lambda}\mu\tilde{\mu}$-calculus terms as an abstract machine, so instead added the constructs $\{\cdot\}$ and $!$ to $\bar{\lambda}\mu\tilde{\mu}$.

Other constructs could've been used, such as a special value $\bar{\lambda}!.M$ for $\{M\}$ and special context $! \cdot e$ for $[\ ]!$. We could've also used $v!$ as a value, with no added context $! \circ e$. We decided to keep the syntax as close to CBPV as possible to be faithful to it's origin and purpose, while still utilising the context $! \cdot e$ in order to specify normal forms without too much deep-diving into a term. If we were to preserve the original syntax exactly, a normal form would look like $\langle x! \mid e\rangle$, $\langle \bar{\lambda}x.v! \mid e\rangle$, etc. To us, examining the structure of a value (or context) in $\bar{\lambda}\mu\tilde{\mu}$ should be done at the top-most level, as all (other) constructs are functions or variables.

In order for simulation of CBPV reduction frames, we must have the reduction frame $\langle [\ ] \mid e\rangle$ in our augmented reduction of $\bar{\lambda}\mu\tilde{\mu}$. In our opinion, this goes against the intended use for an abstract machine, which should operate at the top-most level. We could remove this reduction frame, and instead re-define $\rightsquigarrow_{(\mu)}$ to have priority over all other reduction. This gives the reduction

$$\begin{aligned}\llbracket(...((MV_1)V_2)...)V_n\rrbracket &= \mu\alpha.\langle\mu\beta_{n-1}.\langle ... \mid \llbracket V_{n-1}\rrbracket \cdot \beta_{n-1}\rangle \mid \llbracket V_n\rrbracket \cdot \alpha\rangle \\ &\rightsquigarrow_{(\mu)}{}^{n-1}\ \mu\alpha.\langle\llbracket M\rrbracket \mid \llbracket V_1\rrbracket \cdot ... \cdot \llbracket V_n\rrbracket \cdot \alpha\rangle\end{aligned}$$

which is exactly how the computation $(...((MV_1)V_2)...)V_n$ would run in a CBPV stack machine. But

this gives a problem with the following reduction simulation:

$$(((\lambda x.\texttt{return}\ x)y_1)y_2)y_3 \rightsquigarrow ((\texttt{return}\ y_1)y_2)y_3$$

$$\begin{aligned}\llbracket(((\lambda x.\texttt{return}\ x)y_1)y_2)y_3\rrbracket &= \mu\alpha.\left\langle\mu\beta_2.\left\langle\mu\beta_1.\left\langle\bar{\lambda}x.x \mid y_1\cdot\beta_1\right\rangle \mid y_2\cdot\beta_2\right\rangle \mid y_3\cdot\alpha\right\rangle\\ &\rightsquigarrow_{(\mu)}{}^2\ \mu\alpha.\left\langle\bar{\lambda}x.x \mid y_1\cdot y_2\cdot y_3\cdot\alpha\right\rangle\\ &\rightsquigarrow_{\beta}\ \mu\alpha.\left\langle y_1 \mid y_2\cdot y_2\cdot\alpha\right\rangle\end{aligned}$$

But this is not the interpretation of $((\texttt{return}\ y_1)y_2)y_3$:

$$\llbracket((\texttt{return}\ y_1)y_2)y_3\rrbracket = \mu\alpha.\left\langle\mu\beta_1.\left\langle y_1 \mid y_2\cdot\beta_1\right\rangle \mid y_3\cdot\alpha\right\rangle$$

The actual interpretation does reduce under one $\rightsquigarrow_{(\mu)}$ step to the desired result, but this is a property weaker than simulation.

## 10.2 Future work

Formalising the backwards simulation of the CBPV interpretation (Proposition 9.8) would be the immediate next step. Utilising the Coq [14] formalisation of CBPV [13], we could formalise this work in Coq and prove our work in this mechanism.

An interesting question is the role of $\lambda\mu$ in this interpretation. $\bar{\lambda}\mu\tilde{\mu}$ can be thought of as an extension to $\lambda\mu$ (in particular $\bar{\lambda}\mu$, which is isomorphic to $\lambda\mu$ [1]). This paper doesn't go into detail about the use of the $\lambda\mu$-calculus in constructing the $\bar{\lambda}\mu\tilde{\mu}$-calculus, but one could investigate what constructs from the $\lambda\mu$ calculus could model different CBPV constructs, and compare the difference and similarities of that model with this $\bar{\lambda}\mu\tilde{\mu}$ model.

This work also opens up investigation into a categorical treatment of $\bar{\lambda}\mu\tilde{\mu}$ (and $\lambda\mu$ if the previous work is done). The work done with CBPV in category theory can be applied to $\bar{\lambda}\mu\tilde{\mu}$ through this translation, and a comparison with the current categorical semantics of $\lambda\mu$ [15] could also be investigated.

# Bibliography


[1] P.-L. Curien and H. Herbelin. The Duality of Computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243. ACM, 2000. doi: 10.1145/351240.351262. pages 1, 2, 21, 46

[2] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48959-7. pages 1, 29

[3] José Espírito Santo. The polarized $\lambda$-calculus. *Electr. Notes Theor. Comput. Sci.*, 332:149–168, 2016. pages 1

[4] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proceedings of 3rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'92)*, volume 624, pages 190–201, 1992. doi: 10.1007/BFb0013061. pages 2, 18, 21

[5] Pierre-Louis Curien and Guillaume Munch-Maccagnoni. The duality of computation under focus, 2010. pages 2, 36

[6] S. van Bakel and P. Lescanne. Computation with Classical Sequents. *Mathematical Structures in Computer Science*, 18:555–609, 2008. doi: 10.1017/S0960129508006762. pages 2, 21, 25

[7] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, USA, 2001. ISBN 1402017308. pages 2, 37

[8] G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39(2): 176–210 and 405–431, 1935. pages 3, 5, 8

[9] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. ISSN 00029327, 10806377. URL http://www.jstor.org/stable/2371045. pages 9

[10] William A. Howard. The formulae-as-types notion of construction. 1969. transcribed by Armando B. Matos, 2017. pages 9, 13

[11] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5 (2):56–68, 1940. ISSN 00224812. URL http://www.jstor.org/stable/2266170. pages 11

[12] Philippe de Groote. On the relation between the lambda-mu-calculus and the syntactic theory of sequential control. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, LPAR '94, page 31–43, Berlin, Heidelberg, 1994. Springer-Verlag. ISBN 3540582169. pages 20

[13] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. Call-by-push-value in coq: Operational, equational, and denotational theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, page 118–131, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362221. doi: 10.1145/3293880.3294097. pages 29, 31, 46

[14] URL https://coq.inria.fr/. pages 46

[15] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001. doi: 10.1017/S096012950000311X. pages 46